API Misuses – A Journey Along Their Causes and Prevention to Automated Techniques for Detection and Repair

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Sebastian Nielebock

geb. am 13.09.1989 in Wolmirstedt

Gutachterinnen/Gutachter

Prof. Dr. rer. nat. Frank Ortmeier
Prof. Dr. Alexander Serebrenik
Prof. Dr.-Ing. Thomas Thüm

Magdeburg, den 06.10.2025

Nielebock, Sebastian:
API Misuses - A Journey Along Their Causes and Prevention to Automated Techniques for Detection and Repair
Dissertation, Otto-von-Guericke University Magdeburg, 2025.

Abstract

Context A prevalent amount of modern software development relies on reuse of existing software components and functionalities from software libraries or frameworks. Such libraries or frameworks encapsulate expert and domain knowledge (e.g., secure cryptographic algorithms) behind a so-called *Application Programming Interface (API)*. So-called *API developers* design and develop these APIs. This way, other developers – so-called *client developers* – who do not necessarily have detailed expert knowledge can reuse these software components and functionalities via the API. We denote this as *API usage*. When using the API, client developers can apply the API differently than it was expected by the API developers, which can cause a *deviant, negative behavior* of the software. We denote this as *API misuse*. API misuses are known to be prevalent and have severe functional (e.g., software crashes) as well as non-functional (e.g., security issues) implications.

Objective In this thesis, we explore API misuses from three essential perspectives.

First, we aim to contribute an overview of the *root causes* of API misuses, which not only focuses on the root causes but also emphasizes the methodologies and, thus, the scientific validity of these causes. We also relate the root causes to known *prevention mechanisms* (i.e., techniques and processes), determine the sufficiency of research efforts on prevention, and identify research gaps.

Second, we target *improvements on API misuse detectors*, which currently suffer from high false positive rates (i.e., falsely denoting correct API usages as misuses). We aim to leverage changes of API usages in two ways: (1) using them as a basis to detect misuse in the change itself, and (2) reusing previous fix changes of API misuses as template for finding similar misuses.

Third, we aim to explore the ability to use the artifacts from API misuse detection as repair templates *for an Automated Program Repair (APR) technique*.

Method We obtain the overview of root causes and prevention mechanisms by applying two subsequent Systematic Literature Reviews (SLRs) on relevant empirical studies. Afterward, we map the causes, prevention mechanisms, and their applied research methodologies by applying open coding from qualitative research. To summarize the state-of-the-art detection and repair techniques, we also apply SLRs. We target the improvement in detection and the exploration of repair by engineering research, namely, implementing and validating software artifacts. The experiments of these artefacts reuse existing real-world API misuse datasets from research as well as a dataset we collected and preprocessed from open-source projects.

Results Based on the three perspectives, namely *causes & prevention*, *detection*, and *repair*, we summarize four essential contributions.

First, based on our summary of several root causes, we identified research gaps for certain root causes as well as the previously unexplored dependencies among several causes. We also provided a mechanism to describe these dependencies.

Second, using our obtained summary of the prevention mechanisms, we identified research gaps to effectively target the aforementioned root causes.

Third, we found that our both techniques for API misuse detection, namely, using the change to identify misuses as well as applying previous fixes as a detection basis, improve the precision compared to state-of-the-art misuse detectors. This way, the second approach (i.e., using previous fixes) outperforms the first one (i.e., leveraging change information itself).

Fourth, we found that the artifacts from misuse detection also provide promising results toward an API-specific APR, with our approach having conceptual benefits compared to conventional APR techniques.

Conclusion In summary, we provide an overview of the causes & prevention of API misuses and contribute techniques for their automated detection and repair. This way, researchers can better steer their research efforts on API misuses for different steps discussed in this thesis and build on our software artifacts and results. For practitioners, we provide valuable insights on applicable techniques and processes for API usages, which can be included in the software development as well as the education.

Keywords Application Programming Interface, Misuses, Software Reuse, Root Causes, Prevention Mechanisms, Automated Misuse Detection, Automated Misuse Repair

Zusammenfassung

Kontext Ein umfassender Anteil der modernen Softwareentwicklung basiert auf der Wiederverwendung von Softwarekomponenten und -funktionalitäten aus Softwarebibliotheken oder -frameworks. Solche Bibliotheken oder Frameworks kapseln Experten- und Domänenwissen (bspw. sichere Kryptografiealgorithmen) in sogenannten *Application Programming Interface (API), dt. Programmierschnittstellen*. Sogenannte *API Entwickelnde* entwerfen und entwickeln diese APIs. Somit können durch die API andere Entwickelnde – sogenannte *Client Entwickelnde* –, die nicht notwendigerweise detailliertes Expertenwissen besitzen, die Softwarekomponenten und -funktionalitäten wiederverwenden. Wir bezeichnen dies als *API Verwendung*. Beim Verwenden der API können Client Entwickelnde die API anderes benutzen, als dies durch die API Entwickelnden erwartet wurde, was zu einem *abweichenden, negativen Verhalten* der Software führen kann. Wir bezeichnen dies als *API Fehlverwendung*. API Fehlverwendungen sind ein prävalentes Problem und verursachen schwere funktionale (bspw. Softwareabstürze) als auch nicht-funktionale (bspw. IT-Sicherheitslücken) Probleme.

Zielstellung In dieser Dissertation, erforschen wir API Fehlverwendungen aus drei essentiellen Perspektiven.

Erstens, zielen wir darauf ab einen Überblick über die *Kernursachen* von API Fehlverwendungen beizutragen, welcher nicht nur auf die Kernursachen an sich fokussiert, sondern auch die Methodiken und somit die wissenschaftliche Validität dieser Kernursachen mit einbezieht. Zudem verknüpfen wir die Kernursachen mit bekannten *Präventionsmechanismen* (d.h. Techniken und Prozessen), bestimmen die Hinlänglichkeit des Forschungsaufwands für Präventionen und identifizieren Forschungslücken.

Zweitens, streben wir *Verbesserungen von API Fehlverwendungsdetektoren* an, da jene aktuell hohe Falsch-Positiv-Raten aufweisen (d.h., falsche Erkennung von korrekten API Verwendungen als Fehlverwendungen). Wir beabsichtigen dabei Änderungen von API Verwendungen zweierlei einzusetzen: (1) Nutzen der Änderungen selbst um darin Fehlverwendungen zu erkennen und (2) Wiederverwenden von vorherigen korrigierenden Änderungen von API Fehlverwendungen als Vorlage, um ähnliche Fehlverwendungen zu finden.

Drittens, evaluieren wir die Möglichkeit die Artefakte aus der API Fehlverwendungsdetektion als Reparaturvorlage *für eine automatisierte Programmreparaturtechnik* zu nutzen.

Methodik Wir ermitteln den Überblick von Kernproblemen und Präventionsmechanismen durch die Nutzung zweier aufeinanderfolgender systematischer Literaturrecherchen von relevanten empirischen Studien. Danach verbinden wir die Kernursachen, die Präventionsmechanismen sowie deren zugehörige wissenschaftliche Methoden durch Anwendung von Open Coding Methodiken aus der qualitativen Forschung. Zur Zusammenfassung des Stands der Forschung von Detektions- und Reparaturtechniken nutzen wir ebenfalls systematische Literaturrecherchen. Die angestrebte Verbesserung der Detektion sowie die Exploration von Reparaturtechniken beabsichtigen wir mit Methoden der Ingenieurwissenschaften, konkret der Implementierung und Validierung von Softwareartefakten durchzuführen. Im Rahmen der experimentellen Analyse dieser Artefakte nutzen wir bestehende Forschungsdatensätze von realen API Fehlverwendungen sowie einem von uns gesammelten und aufbereiten Datensatz von Open Source Projekten.

Ergebnisse Basierend auf den drei Perspektiven *Ursachen & Präventionen*, *Detektion* und *Reparatur*, können wir vier essentielle Beiträge zusammenfassen.

Erstens, basierend auf unserer Übersicht zahlreicher Kernursachen, identifizierten wir Forschungslücken für bestimmte Ursachen als auch für bisher nicht erforschte Abhängigkeiten zwischen zahlreichen Kernursachen. Zudem stellen wir einen Mechanismus bereit um diese Abhängigkeiten zu beschreiben.

Zweitens, mittels unserer Übersicht über Präventionsmechanismen identifizierten Forschungslücken jener, um die zuvor genannten Kernursachen effektiv besser zu adressieren.

Drittens, ermittelten wir das unsere beiden Techniken zur API Fehlverwendungsdetektion, konkret, Änderungen zur Identifikation von Fehlverwendungen nutzen, als auch die Verwendung vorheriger Korrekturen als Detektionsgrundlage, die Präzision gegenüber aktuellen Fehlverwendungsdetektoren verbessern. Dabei übertrifft der zweite Ansatz (d.h. Verwenden früherer Korrekturen) den ersten (d.h. Nutzen der Änderung selbst).

Viertens, fanden wir heraus, dass Artefakte der Fehlverwendungsdetektion auch vielversprechende Ergebnisse für eine API-spezifische, automatische Programmreparatur ermöglichen, wobei unser Ansatz konzeptionelle Vorteile gegenüber konventionellen automatischen Programmreparaturtechniken besitzt.

Fazit Zusammenfassend bieten wir eine Übersicht der Kernursachen & Präventionen von API Fehlverwendungen und leisten Beiträge in Form von automatisierten Techniken zur Detektion und Reparatur jener Fehlverwendungen. Dadurch können Forschende ihren Forschungsaufwand bzgl. API Fehlverwendungen anhand der in dieser Dissertation diskutierten Schritte besser steuern und unsere Softwareartefakte und Ergebnisse als Grundlage weiterer Forschung nutzen. Für Anwendende bieten wir wertvolle Erkenntnisse bzgl. anwendbarer Techniken und Prozesse für API Verwendungen, die in die Softwareentwicklung und Ausbildung einfließen können.

Schlüsselwörter Programmierschnittstellen, Fehlverwendungen, Softwarewiederverwendung, Kernursachen, Präventionsmechanismen, Automatisierte Fehlverwendungsdetektion, Automatisierte Fehlverwendungsreparatur

Contents

List of Figures

List of Tables

Code Listing

List of Acronyms

AL Active Learning
API Application Programming Interface
APR Automated Program Repair
ASAP-Repair API-Specific Automated Program Repair
AST Abstract Syntax Tree
AUG API Usage Graph
ChaRLI Change Rule Inference
CFG Control Flow Graph
CI Continuous Integration
DCA Dynamic Code Analysis
DL Deep Learning
FGM Frequent Graph Mining
FPM Frequent Pattern Mining
FSM Frequent Sequence Mining
GED Graph Edit Distance
Groum Graph-based Object Usage Model
HMM Hidden Markov Model
LLM Large Language Model
MCS Maximum Common Subgraph
MSR Mining Software Repositories
NLP Natural Language Processing
PDG Program Dependence Graph
RAIX Relevant API Information Extractor
RuDetect Change Rule-based API Misuse Detection
SBFL Spectrum-based fault localization
SCA Static Code Analysis
SLR Systematic Literature Review
VCS Version Control System

1 Introduction

This chapter introduces the topic of *Application Programming Interface (API)* usages and misuses (cf. Section 1.1), our three main research questions (cf. Section 1.2), our contributions to these research questions (cf. Section 1.3), and the general structure of the thesis (cf. Section 1.4).

1.1. API Usage and Its Problems

Modern software development heavily relies on the reuse of already available functionalities. Such reuse ranges from simple code copying from previous projects or online discussion forums such as StackOverflow[1] [ZUR+18] up to using specialized libraries or frameworks for specific tasks, for instance, to create a graphical user interface (GUI) using a framework such as JavaFX[2]. Past research analyzed the prevalence of reuse from (third-party) libraries. For instance, Wang et al. [WGMC15] found that more than 60% of 105,299 Android apps had more than 1,024 calls to libraries. In another study on 806 Java projects, the authors found that only 74 projects (9.2%) used *no* libraries in their methods [WCH+20].

Developers enable library reuse by providing *APIs*. APIs describe how methods, data fields, and exception handlers [ZM19] – in general, ***API elements*** – implemented within a library or framework are externally accessible to other developers. When reusing an API element, we denote this as an ***API usage***. API usage, a form of code reuse, aims to save programming time and costs, reduces the burden of testing the underlying implementation, and abstracts away most of the subtle details of the underlying implementation. Using this abstraction and the concluding separation of concerns, the code becomes more maintainable [Som18, p. 491pp] [WTH+24, p. 4-7p].

```
1  import android.widget.Button;
2  //[...]
3  public class AndroidTest {
4      //[...]
5      public void handleButton(Button button){
6          //[...]
7          button.callOnClick();
8          //[...]
9      }
10     //[...]
11 }
```

Code Listing 1.1: Example of an API Usage for the `android.widget.Button`-class of the Android framework.

We provide an example of an API usage in Code Listing 1.1 for the Java programming language and the Android framework, a prominent mobile operating system[3]. In detail, the

[1] https://stackoverflow.com/ last accessed: 2024/10/09
[2] https://openjfx.io/ last accessed: 2024/10/09
[3] https://developer.android.com/ last accessed: 2024/10/10

`android.widget.Button`-API is reused through the `import` statement in the class and the call of the API method `callOnClick`, which triggers listeners attached to the click event of the `button` object.

Users of APIs assume that these are implemented by experts in the library's domain (e.g., GUI or mobile experts) or involve domain experts in the API design, and thus, APIs enable the reusability of this expert knowledge for other developers [MKA+18]. However, in some domains, such as for cryptographic APIs, it was observed that their developers are not always experts [RRS23]. We refer to these developers, who produce the API as ***API developers*** and to developers who use APIs in their code as ***client developers***.

Client developers do not always apply an API as it is intended by the API developers. For instance, the API usage in Code Listing 1.1 will crash if the `callOnClick`-method is called using the Android framework with API level 15 or lower[4]. Another example is the usage of a cryptographic API that causes privacy issues, for example, by using an outdated and easily decodable MD5 hash function instead of a more secure SHA-256 function [KSA+21]. Both examples describe a ***deviant API usage*** from what is expected by API developers that causes a ***negative software behavior***. We define this as an ***API misuse*** [NK11]. This definition is similar to the one by Schlichtig et al. [SSNB22] based on a literature review, namely, that an API misuse is a violation of a documented or implicit API usage constraint, emphasizing that such violation causes errors, misbehavior, or similar issues.

As demonstrated before, the negative behavior can imply software crashes and performance issues but also probably not easily observable security or usability issues. Thus, API misuses have implications for both functional and non-functional requirements. Thus, we state that API misuses can have severe implications for the end users of the software.

In the past, many studies observed the prevalence of API misuses in practice. For instance, Zhong and Su [ZS15] found that half of the source files edited in fixes within five open-source projects required at least one API-specific change. In another study, Amann et al. [ANN+16] found a proportion between 3.9%-62.5% (overall proportion is 10.4%[5]) of API misuses. Zhang et al. [ZUR+18] measured that 31% of code samples provided as solutions from StackOverflow denote possible API misuses. Due to this prevalence, namely, the amount of reuse of code via APIs and the number of API misuses, a lot of research (i.e., in terms of different research groups and techniques) on automated API misuse detection has been conducted [WN05, WZL07, NNP+09b, LW09, PG09, MBM10, Lin15, MCJ17, ANN+19a, KSA+21, LMC+21, YRW22, WZ23].

APIs and their misuses involve different stakeholders (i.e., API and client developers) with different domain knowledge (e.g., cryptography or mobile experts). APIs abstract detailed expert knowledge as reusable interfaces and leverage artifacts beyond source code, such as documentation or Q&A discussion forums. This way, the knowledge, processes, and techniques on APIs differ from those of typical code reuse and bugs. Therefore, research focuses on APIs to obtain better-suited support in software development. Despite the tremendous effort in this area, current research on API misuses struggles with different challenges, with three of them motivating this thesis:

[4] https://developer.android.com/reference/android/view/View#callOnClick() last accessed: 2024/10/10

[5] These numbers diverge from the reported percentage by Amann et al. [ANN+16], since we could not replicate their values and re-computed them based on Table 1 in this paper.

C&P: Lack of Knowledge on API Misuse Root Causes and their Prevention: While much research was conducted on API misuse detection, these works rarely analyzed the root causes of misuses and how to prevent them. Currently, there only exist restricted summaries on root causes. A systematic overview of the most prevalent root causes would greatly improve the work on misuse prevention. It also remains uncertain to which degree state-of-the-art prevention mechanisms target those root causes effectively.

D: Imprecise API Misuse Detection: Current misuse detectors struggle due to imprecise detection, namely, they report correct usages as API misuses (i.e., so-called *false positive* detection), lowering their acceptance in practice due to many 'false alarms' [LW09, ANN+19a].

R: Limited usage of API-specific knowledge for Automated API Misuses Repair: Assuming a precise misuse detection, and considering the tremendous research progress in Automated Program Repair (APR) [Mon18a, LGPR19, KMSH21], a logical next step is an API-specific APR technique targeting API misuses.

1.2. Research Questions on API Misuses

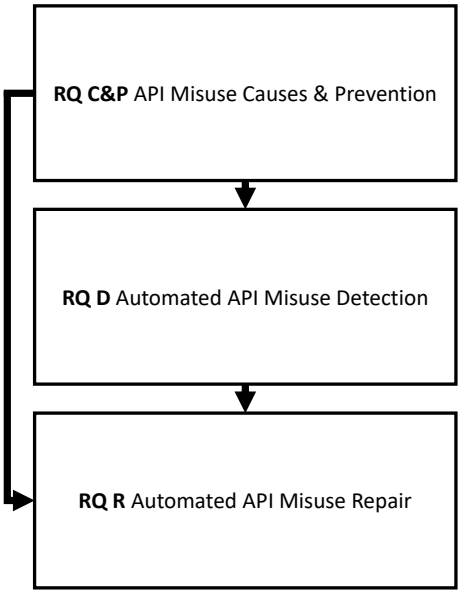


Figure 1.1.: Structure of the Main Research Questions

In this thesis, we summarize the three challenges as the three following research topics from which we derive the research questions:

C&P API Misuse Causes & Prevention

D Automated API Misuse Detection

R Towards Automated API Misuse Repair

By addressing these topics, we target API misuses in different stages during development, namely, at the time of introducing misuses (i.e., C&P), at the time misuses are present in the code (i.e., D), and at the time misuses should be fixed (i.e., R).

In the following sections, we describe and justify the questions in detail concerning the state-of-the-art. Figure 1.1 depicts an abstract view of how these research questions intertwine and which will be subsequently and refined and used as guidance among this thesis.

1.2.1. Research Question C&P - API Misuse Causes & Prevention

We consider an API misuse as a type of a *code defect*. We refer to Zeller [Zel06], who defined that *"[a code]* defect *is a piece of code that can cause an infection"* (cf. [Zel06, Sec. 1.2, p. 3]). Further, Zeller denoted an *infection* as a deviant behavior of the software, which eventually causes a *failure*. A *failure* itself is the negative behavior observed from the software [Zel06] [WTH+24, p. 12-3].

Thus, we conjecture that API misuse prevention is closely related to the more general term of *defect prevention*, which is considered useful in practice [MJHS90], for instance, by

conducting code reviews [BB13, SSC+18] or applying DevOps concepts [LRK+19, EH23] to ensure software quality [WTH+24, p. 12-15].

Usually, prevention encompasses an analysis of the root causes of previous defects. As known from quality management [KCT12] and particularly software quality management [WTH+24, p. 12-1pp], these root causes include, for instance, the quality of the input and output of a process, the tools used, methods applied, and the people involved. Root causes are determined by techniques such as defect causal analysis [Car98, KCT12] or techniques from root cause analysis [WTH+24, p. 18-16p], including Fault Tree Analysis [OS07] or cause-effect-diagrams like the Ishikawa diagram [Ish90].

Typical defect prevention applies actions to mitigate or eradicate root causes. In practice, these actions decrease the defect rate by more than 50% [KCT12]. The actions involve various aspects and steps of the software development cycle, such as requirements engineering [JBR99, Som18], requirements traceability [FGO17], test-driven development [Bec02], agile software development [Mar03, Som18], software documentation [WTH+24, p. 14-8p], modern code review [BB13, SSC+18], programming education [PSM+07, MRF19], or applying best practices for software development [GHJV94, Mar13, FBB+14].

Nevertheless, the previously discussed specialties of APIs are not covered by these general processes and techniques. Thus, we focus on *API misuse causes & prevention*, which we denote as the steps before API misuses occur and thus target the *root causes* of misuses [Rob09, HL11, MKA+18] as well as techniques and processes [BFHM12, NAP18, RTP19, TCK21, PDHR23] to avoid them. Knowledge of root causes and mechanisms targeting these causes and thus preventing API misuses in the first place reduces later efforts for detection and repair.

No Overview on API Misuse Root Causes: In contrast to typical defect prevention, research on root causes of API misuses lacks – to the best of our knowledge – a systematic overview despite many individual studies [SM08, Rob09, HL11, ZER11, NKMB16, QLL16, ABF+17, SMAR17, ANBL18, MSS18, MKA+18, PHR19, GALIF20, LS20, ZHKG20, WHH+24]. We formulate this as a challenge to *derive a systematic overview of the applied methodologies and most prevalent root causes of API misuses.*

No Overview and Unclear Effect of API Misuse Prevention: For API misuse prevention, *it remains unclear whether and to what degree state-of-the-art defect prevention mechanisms target the root causes.* This is due to two reasons: First, because - to the best of our knowledge - no systematic overview of prevention mechanisms exists, and second because we do not know the most prevalent root causes to be mitigated or eradicated.

Thus, we summarize these challenges as the following research question:

Research Question C&P - Causes & Prevention

RQ C&P (Causes & Prevention): *Are root causes of API misuses sufficiently targeted by API misuses prevention mechanisms?*

1.2.2. Research Question D - Automated API Misuse Detection

Reusing the definition by Zeller [Zel06] on code defects, Zeller also denoted that a caused infection can spread among the software, and thus, the location of the observable failure is not directly the location of the code defect. Code defect detection encompasses *recognizing* that a defect exists (e.g., observing a negative behavior) and *locating* the defect (i.e., identifying the source code lines triggering the negative behavior) [Zel06, WGL+16].

General code defect detection includes several techniques and processes, such as Static Code Analysis (SCA) and Dynamic Code Analysis (DCA), code inspection, verification, or testing [WTH+24]. Thus, research suggested and evaluated various techniques and processes to support these steps [RAT+06]. While we detail the range of code defect detectors in Section 3.1.2, it is known from past research that an effective and efficient application of them depends on several factors, such as the persons applying these techniques, the code at hand to be tested, as well as the type of code defect [RAT+06].

This last factor explains why many researchers developed automated techniques to specifically detect API misuses [WN05, LZ05, WZL07, NNP+09b, MCJ17, ANN+19b, KL21, ZCSZ21, LCP+21, NVN20, KSA+21]. *API misuse detection* describes the situation in which a misuse is likely present in the code and which should therefore be recognized and located by a technique or a process [RBK+13, ANN+19a, EHJ+21]. Without detection of API misuses, one cannot conduct steps for repairing them.

A common notion of these API misuse detectors is to infer a *specification* describing a *valid* API usage and check whether an API usage at hand satisfies this specification. Specifications are hand-made [KSA+21] or inferred automatically, for instance, mining frequent API usage as patterns [WN05, LZ05, NNP+09b, WZ11, ANN+19a, KL21] applying Frequent Pattern Mining (FPM). We refer to these techniques as *pattern-based API misuse detection*. Manual or automated inference assumes the ability to describe correct API usage. For the manual case, the concrete assumption is that an expert can sufficiently define a valid API usage. In the automated case using FPM, the latent assumption is that frequently re-occurring API usage is likely to be correct.

High False Positive Rate of API Misuse Detection: However, research had shown that both techniques *lack sufficient precision* (i.e., the proportion of correctly reported misuses from all reported misuses) *when detecting API misuses* [LHX+16, ANN+19a]. From SCA, we know that a low precision causes many 'false alarms' and, thus, reduces the applicability of defect detection [CM04, JSMHB13, SAE+18].

Accordingly, research on API misuse detection aims to increase the precision while keeping a significantly large recall (i.e., the proportion of detected misuses among all misuses). Much effort has been put into improving the mining [LZ05, NNP+09b, TX09a, ZXZ+09, NK11, PG12, ÇM18, KL21, WZ23, WXQ23] and post-pattern mining process (e.g., filtering or ranking of patterns) [ECC01, WN05, LZ05, PG09, DLMK10, ÇM18, ANN+19b, WXQ23]. Still, these approaches do not achieve a sufficient precision value for practical usage (cf. Section 5.2.2).

Unrealistic Use Cases for API Misuse Detection: State-of-the-art API misuse detectors often *lack realistic use cases for practical application*. Particularly, manually writing correct specifications is tedious, error-prone, and thus less applicable in practice [HSSA16, LHX+16]. For pattern-based approaches, research usually collects donor code, namely, code samples from which patterns are mined by explicitly searching examples of the misused API. However, in a practical use case, the *misused API* is not known. Thus, both processes limit the scalability of API misuse detectors in practice.

We target both problems, namely, high false positive rate and unrealistic use cases, by *enhancing the donor code for pattern-based API misuse detection* as well as *introducing change-based API misuse detection*.

Thus, we summarize the research question as follows:

Research Question D - Detection

RQ D (Detection): *How can we improve the precision of state-of-the-art API misuse detectors within a realistic software development process?*

1.2.3. Research Question R - Automated API Misuse Repair

According to Le Goues et al. [LGPR19], APR involves three steps:

(1) detecting a code defect based on a specification (e.g., a test suite),

(2) synthesizing a patched version of the code, and

(3) validate the patched code (e.g., by applying the aforementioned specification).

Since steps (1) and (3) are undecidable, the general problem of APR is undecidable as well [LGPR19, NL22]. Nevertheless, developers are able to fix defects manually in particular cases and for specific defect types. When restricting the automated approach to certain cases, APR has been proven applicable. This observation motivates a large research community to actively work on techniques for APR [Mon18a, LGPR19], having an ongoing workshop[6] as well as a community website informing on their research.[7]

Since no general solution for APR exists, concrete techniques will always adapt to certain use cases and defect types [NL22]. Thus, we focus on automated techniques to fix a successfully detected API misuse by permanently mitigating its negative behavior or completely removing it [Nie17, KMSH21]. We denote this as an *API misuse repair*. In this thesis, we focus on automated techniques for repair. While Kechagia et al. [KMSH21] have demonstrated that state-of-the-art APR techniques can, to some degree, target API misuses, these techniques miss *the potential to leverage the special characteristics of API misuses*.

Changing Specifications for API Misuse Repair: We focus on one of many challenges discussed by Le Goues et al. [LGFW13, LGPR19] of APR research. This challenge describes that some patch synthesis techniques require donor code. These techniques need to find ways to *deal with changing code samples* [LGFW13]. This is particularly true for APIs since API developers maintain and update code libraries, and so the API, as well as their latent usage specifications. Particularly, *data-driven-based approaches to infer API usage patterns provide the potential to fill this gap* as they mine patterns from related code samples.

Not Using Historical API Misuse Fixes: Moreover, some APR approaches had success by *leveraging information from past fixes* in their repair strategies [LLLG16, JXZ+18]. Since this notion aligns with the idea of leveraging change information of API usage, we hypothesize a similar effect when using change information of past fixes of API misuses.

We summarize this by the following research question:

Research Question R - Repair

RQ R (Repair): *Does API-specific information support automated API misuse repair?*

[6] https://program-repair.org/workshop/ last accessed: 2025/02/28
[7] https://program-repair.org/ last accessed: 2025/02/28

1.3. Contributions of the Thesis

We describe the contributions of the aforementioned research questions on causes & prevention (C&P), detection (D), and repair (R) in the following.

1.3.1. Contributions for RQ C&P - Causes & Prevention

Using a Systematic Literature Review (SLR) methodology [KC07, Woh14], we collected 65 studies on root causes of API misuses and 411 publications on prevention mechanisms that target these causes. We summarized and interrelated the studies using qualitative methods such as open coding [Fli14]. This summary provides a *new comprehensive and interrelated overview of the root causes of API misuses as well as the scientific methodologies to collect them* as well as *an overview of state-of-the-art prevention mechanisms* targeting these root causes. Based on these results, we analyzed the sufficiency of prevention mechanisms for API misuse root causes. We summarize our *main contributions* as follows:

(1) We present a *diverse set of API misuse root causes with eleven different categories with 44 sub-root causes*. Moreover, we suggest hypotheses on their interdependencies, namely, whether root causes trigger others using so-called *views*, and provide three examples using the developer, process, and technical perspectives.

(2) We present the *typical structure of the studies on API misuse root cause analysis* and show that a majority of root causes were evaluated using *qualitative methods*.

(3) We provide an *overview of prevention mechanisms* targeting API misuse root causes, which we classify as *recommendations and automated support*.

(4) We discuss that research on prevention mechanisms has a *focus on single root causes* and *typically validates automated support mechanisms*, while a *majority of recommendations remain non-validated*.

We conclude that due to the small number of quantitative studies on different kinds of root causes as well as the variety of different root causes, current research can hardly judge their prevalence. Therefore, *further research needs to conduct more quantitative studies on API misuse root causes* to provide these insights. While there is a focus on the client side of root causes, *more research has to be conducted on the API developer side*. Such research results would allow judging on possible support for API developers, which would boost API quality and, thus, its usability. Based on the quantification of root causes, current and future prevention mechanisms have to validate whether they *target the most prevalent root causes* or whether they *target the root causes triggering the most severe API misuses*. While many prevention mechanisms are present, they typically focus on single root causes and on automated support rather than recommendations. However, recommendations can be more adaptable in practical scenarios (e.g., adapting a software development process). Thus, future work should consider these research gaps.

These results are valuable for practitioners and software engineering researchers alike. Developers in practice can use this body of knowledge on root causes to identify and mitigate them in their daily practice by using well-validated prevention mechanisms. Researchers benefit from these contributions by directing their future work according to studies quantifying the influence of different root causes and observing the perspective of API developers. For replication, we provide our data and scripts.[8]

[8] http://doi.org/10.5281/zenodo.15594600

1.3.2. Contributions for RQ D - Detection

We target the issues of low precision and missing applicability to realistic software engineering processes of state-of-the-art misuse detectors with two main ideas:

(1) We provide concepts and software artifacts to *improve pattern-based API misuse detection*. First, we present our artifact named ***Relevant API Information Extractor (RAIX)***, which extracts API-specific information *leveraging API changes using the Version Control System (VCS)* of a potentially misuse-containing client project. Second, we apply *search and filter strategies to find code samples* serving as donor code for the inference of API usage patterns as specifications. This way, we expect it to seamlessly fit into a realistic development process. Since we hypothesize the code samples to refer to the APIs currently changed in the client project, we expect the pattern-based detectors to use these inferred patterns to more precisely detect API misuses. We evaluated both techniques on real-world API misuse datasets.

Our *main findings* were that API changes from *VCSs are a valuable source* even though for efficient, practical usage, we require further selection strategies. When using search and filter strategies, we found that *previous knowledge of a potentially misused API is not necessary to find sufficient code examples for pattern mining*, and we observed a *positive impact on the pattern quality, especially when using method-level filtering of API usages*. Moreover, we obtained a *slight improvement in the precision and recall for pattern-based API misuse detection*.

(2) We introduce the *concept of change rules*, representing previous API misuse fixes by which similar API misuses can be detected. These *change rules are automatically obtained from previous API misuse fixes by a technique named* ***Change Rule Inference (ChaRLI)***. We present the concept of *using these change rules to detect API misuses* with a technique named ***Change Rule-based API Misuse Detection (RuDetect)***. We provide an *evaluation of different settings of this change rule-based misuse detection* and in *comparison to a pattern-based misuse detector*.

In detail, we obtained the best fitting *settings to apply change rules for API misuse detection based on different inference techniques and graph similarity metrics*, and we observed *a higher precision when using change rule-based misuse detection compared to pattern-based misuse detection*. Regarding the *recall*, we observed a *dependency on the training dataset* from which change rules were inferred.

Thus, we conclude that both variants are valuable technique, even though further steps for practical application are necessary. For instance, we can improve pattern-based misuse detection by more advanced search and filter strategies to further improve precision. Change rule-based detection require steps to control the training data, and thus, recall of the detection. Both techniques represent a step towards practical API misuse detection, and thus an improvement to the state-of-the-art.

These results provide software engineering researchers insights on different filter and search strategies for pattern-based misuse detection as well as a further technique leveraging previous fixes, based on which further improvements can be validated. Moreover, we provide a replication package of our results[8]. For practitioners, the results provide a first step towards transferring misuse detection from research into practice.

1.3.3. Contributions for RQ R - Repair

Current APR techniques do not necessarily target API misuses. Therefore, we provide evidence of how *API-specific information known from API misuse detectors can benefit APR*. In detail, we present a *concept for repairing API misuses named* ***API-Specific Automated Program Repair (ASAP-Repair)***. It leverages an intermediate code representation of API usages named API Usage Graph (AUG) using API usage patterns as well as change rules already used during the previously discussed API misuse detection. ASAP-Repair serves as an intermediate step towards full-fledged API misuse repair by providing insights on a conceptual application of API-specific information and stressing out further research challenges to be targeted to accomplish an effective API misuse repair technique. We validated *different variants using patterns and change rules-based repair concepts using real-world API misuse datasets and quantitatively and qualitatively assessed their produced patches.*

Our *main contributions* are as follows:

(1) *ASAP-Repair is conceptually superior to state-of-the-art APR techniques without requiring test suites for fault localization.*

(2) We demonstrated that *it can repair API misuses on the basis of the intermediate data structure of AUGs*, while *change rules from previous fixes tended to be more promising than API usage patterns.* However, for both techniques, it is *essential to carefully collect sufficient donor code* by which ASAP-Repair generates repair templates for patch generation.

We conclude that ASAP-Repair is a promising technique for APR of API misuses because (1) it is built on top of existing misuse detection data structures, easing the interplay with state-of-the-art misuse detectors, (2) in comparison to existing APR techniques, it does not require the existence of test cases to produce patches, and (3) has been proven to be, in principle, applicable to real-world misuse while still requiring research for a full-fledged APR.

For researchers, ASAP-Repair denotes a first step toward the field of API-specific APR and further research directions. Again, we provide ASAP-Repair and the evaluation as a replication package[8]. While currently, we assess the applicability for practitioners as limited, the intermediate data structure of AUGs can provide support for developers to plan bug fixes and refactoring steps of API misuses.

1.4. Structure of the Thesis

The subsequent section explains the content of the following chapters as well as their overall structure and relations depicted in Figure 1.2.

Chapter 1, Introduction, introduces the problem domain of API usages and misuses as well as derives the main research questions and contributions on causes & prevention, automated detection, and steps towards automated repair of API misuses.

Chapter 2, Problem Analysis, reports a further detailed discussion on the problem domain and refines the main research questions into sub-questions, thus motivating the required background and the applied methodologies to answer the main questions.

Chapter 3, Fundamentals and Background, discusses the general background of software defects with a specific view on the taxonomy and the prevalence of API misuses, code representation, code search, code changes, and Frequent Pattern Mining (FPM).

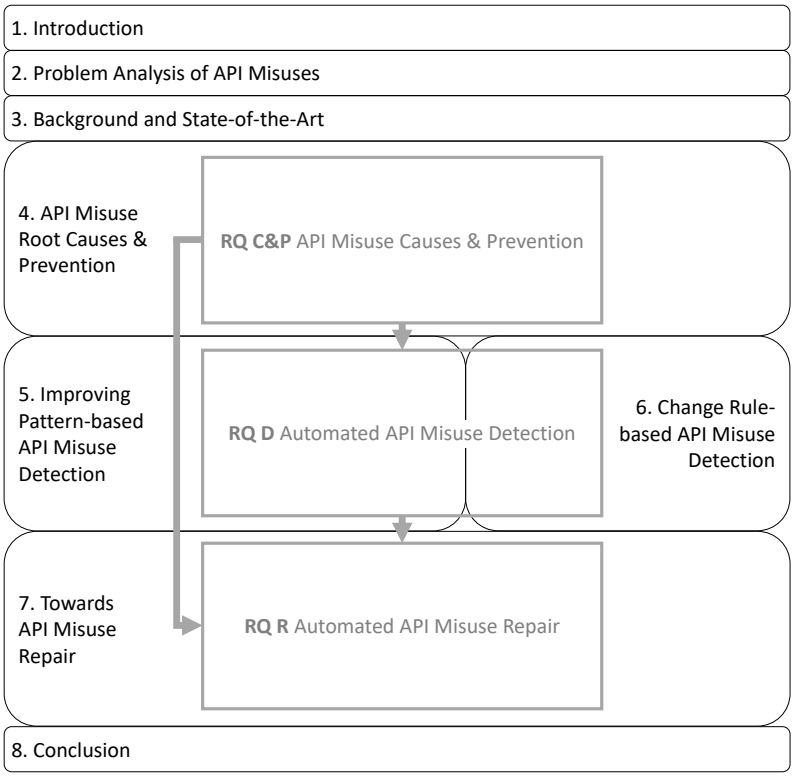


Figure 1.2.: Structure of the Thesis

Chapter 4, Causes & Prevention, targets **RQ C&P** on API misuse root causes & prevention mechanisms with the related sub-research questions discussed in Chapter 2. It requires background on software defects and API misuses from Chapter 3.

Chapter 5, Pattern-based Detection, targets **RQ D**, particularly on limitations and improvements of pattern-based API misuse detectors with the related sub-research questions discussed in Chapter 2. The additional sub-research question on change rule-based API misuse detection is presented in ***Chapter 6, Change Rule-based Detection***. Both require background on software defects, API misuses, code representation, code search, code changes, and FPM from Chapter 3.

Chapter 7, Repair, targets **RQ R** analyzing techniques towards automated API misuse repair with the related sub-research questions discussed in Chapter 2. It requires background on software defects, API misuses, code representation, and code changes from Chapter 3, as well as the API misuse detection presented in Chapters 5 and 6.

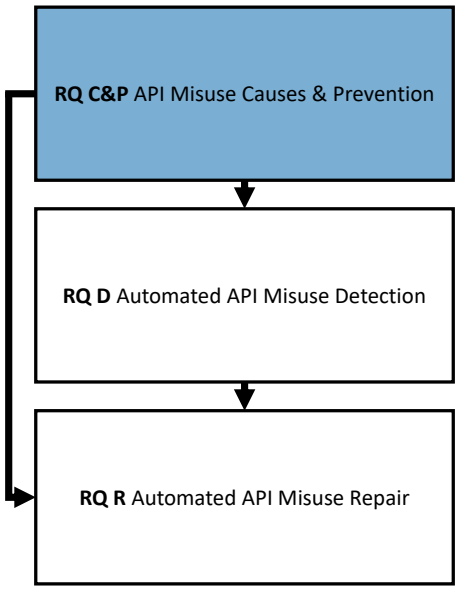
Chapter 8, Conclusion, summarizes and concludes the results and contributions of this work. It gives an overview of possible implications, namely, further research directions as well as steps towards practical applicability on API misuse prevention, detection, and repair.

2

Problem Analysis of API Misuses

This chapter refines the three main research questions from the Introduction (cf. Chapter 1), namely, **RQ C&P** (cf. Section 2.1), **RQ D** (cf. Section 2.2), and **RQ R** (cf. Section 2.3) by a detailed problem analysis. This way, we explain our applied research methodologies and discuss the expected implications of answering the sub-questions as well as their contribution to the main research question. Finally, we give a detailed structure in Section 2.4

2.1. RQ C&P API Misuse Causes & Prevention



For **RQ C&P** targeting *missing knowledge on root causes & their prevention mechanisms of Application Programming Interface (API) misuses*, we ask *whether state-of-the-art prevention mechanisms target root causes sufficiently.* We hypothesize that, first, we do not have a systematic overview of API misuse root causes. Second, we lack a systematic overview of prevention mechanisms targeting these root causes. Third, due to both previous issues, currently, we can hardly judge whether the research on prevention mechanisms targets misuse causes sufficiently. Sufficiency means that prevention mechanisms should have research attention related to the prevalence and severity of their targeted root causes.

We answer the main **RQ C&P** by investigating these three separate sub-problems, namely:

1. Missing Systematic Overview on State-of-the-art Research of API Misuse Root Causes;
2. Missing Systematic Overview on State-of-the-art Research of Prevention Mechanisms for API Misuse Root Causes;
3. Missing Assessment of whether there exists Sufficient Research on Prevention Mechanisms to target all found API Misuse Root Causes.

Missing Systematic Overview on State-of-the-art Research of API Misuse Root Causes. We target the first sub-problem by a summary of API misuse causes. This summary represents a collection and the interrelations of different root causes and can be updated in future work to encompass further insights or new causes. Moreover, it allows us to compare root causes of API misuses to root causes of other kinds of defects, and thus, judging whether related prevention mechanisms are applicable.

We inferred our summary based on the body of knowledge of API misuse causes. For this purpose, we systematically collected, summarized, and interrelated research on API misuse causes. To collect relevant literature, we applied a Systematic Literature Review (SLR) [KC07] and qualitative methods to summarize and interrelate the relevant knowledge on root causes [Fli14]. By using established methodologies from qualitative research, such as *open coding* [Fli14], we systematically structured information on the methods, results, and knowledge gaps of several studies. That means we did not replicate research results but related them to each other, showing common trends, contradictions, as well as limitations of the studies. We provide the documentation of our research steps as well as all results in a replication package[1]. This way, we ensure the falsifiability of the provided summary by allowing independent replication.

We summarize this sub-problem as **RQ C&P-C** (C&P-Root-Causes) and our contribution as follows:

RQ C&P-C (C&P-Root-Causes): ***What are the common root causes of API misuses?***

We provide an overview of the body of knowledge of API misuse causes using SLR. We collected relevant studies and established qualitative methods to summarize and interrelate the methods, results, and knowledge gaps on API misuse causes. We found a diverse set of root causes in terms of numbers (i.e., 11 different main root causes) and variants (e.g., developer- vs. technical- vs. process-related causes). In addition, we elicited trends in the cause analysis (e.g., more qualitative than quantitative studies).

Missing Systematic Overview on State-of-the-art Research of Prevention Mechanisms for API Misuse Root Causes. Next, we target the sub-problem of the missing overview on prevention mechanisms towards the root causes, similar to the method from **RQ C&P-C**. In detail, we applied SLR [KC07] to collect the relevant research on prevention. Prevention encompasses general-purpose techniques like automated tool support (e.g., automated tests), software engineering processes (e.g., code reviews, refactoring) as well as API-specific techniques (e.g., enhancement of API documentation). Based on this collection of literature, we extracted an overview of the relevant research on API misuse prevention using *open coding* [Fli14]. Having this overview, we showcased which kind of prevention is available and which of it is applicable for which root causes. Moreover, the overview guides future research on improving and comparing to the state-of-the-art.

[1] http://doi.org/10.5281/zenodo.15594600

We summarize this as **RQ C&P-P** (C&P-Prevention) formulated as follows:

RQ C&P-P (C&P-Prevention): ***What are the state-of-the-art prevention mechanisms targeting API misuses?***

We give an overview of mechanisms for API misuse prevention, which is based on the overview of the body of knowledge of API misuse root causes using methods from SLR. We applied methods from qualitative research to summarize, relate, and extrapolate methods and results of primary studies on prevention mechanisms. We determined that research on prevention mechanisms is not equally distributed and that recommendations are frequently not validated while automated support typically are.

Missing Assessment of whether there exists Sufficient Research on Prevention Mechanisms to target all found API Misuse Root Causes. We aim to assess the sufficiency of state-of-the-art research on API misuse prevention targeting API misuse root causes in **RQ C&P-S**. Thus, **RQ C&P-S** combines the results of **RQ C&P-C** and **RQ C&P-P**. This combination is necessary since root causes and prevention mechanisms are strongly coupled, meaning that without targeting root causes, prevention mechanisms would be without effect or even harmful. Thus, our results provide evidence (1) which prevention mechanisms are most researched, (2) for which root causes research still lacks effective prevention mechanisms, or for which root causes prevention mechanisms are not effectively applicable. We combined causes and prevention using the concepts and results of preventions' description in their publication. In detail, we applied qualitative methods to match the preventions' description with the previously obtained root causes of API misuses. Based on this matching on API misuse root causes and prevention mechanisms, we extracted and prioritized relevant research directions.

We summarize this sub-research question as follows:

RQ C&P-S (C&P-Sufficiency): ***Does state-of-the-art research on prevention mechanisms sufficiently target root causes of API misuses?***

By combining the results of the bodies of knowledge on API misuse root causes and its prevention mechanisms, we obtained a prioritized research agenda for future directions in root cause analysis and effective prevention mechanisms. Our results indicate that research on prevention mechanisms should focus on evaluating recommendations and only concentrate on specific automated support techniques to target certain root causes.

Structure of RQ C&P As depicted in Figure 2.1, the research question **RQ C&P** is based on the analysis of the state-of-the-art research on misuse causes (i.e., **RQ C&P-C**) and prevention mechanisms (i.e., **RQ C&P-P**) which both provide insights to assess the sufficiency of prevention mechanisms (i.e., **RQ C&P-S**).

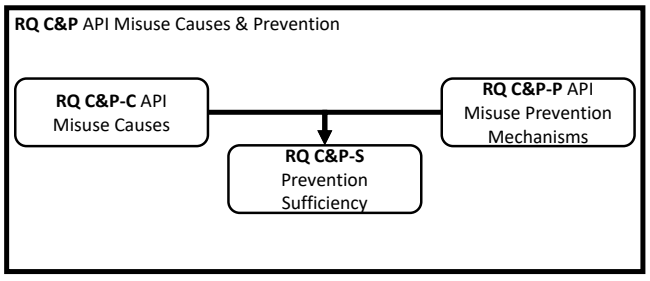
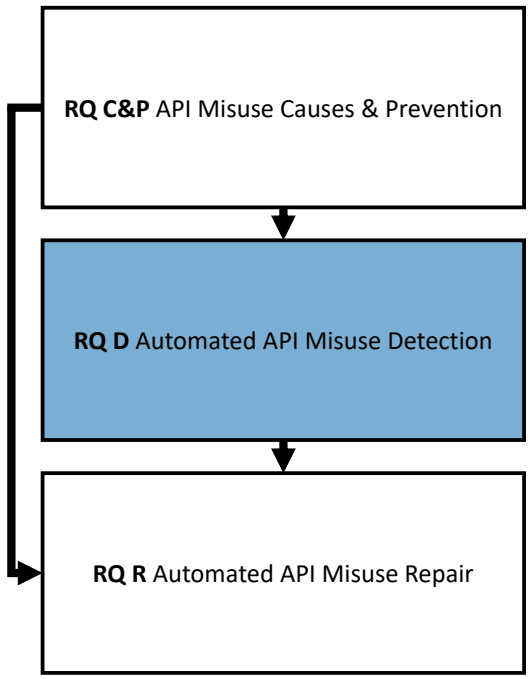


Figure 2.1.: Structure of the Sub-Research Questions of **RQ C&P**

2.2. RQ D Automated API Misuse Detection



While prevention tends to be an effective strategy as it gets down to the root of the negative implications of API misuses, in a realistic software development scenario, perfect prevention is almost impossible, for instance, due to the costs of investments [MJHS90] [WTH+24, p. 12-4]. Particularly, software development heavily relies on human developers who, per se, are not error-free, for instance, influenced by personal mood [Zel06, p. xix]. While automated approaches try to mitigate human errors, the increasing software size and complexity in the last decades [Som18, p. 24], paired with faster development times, restrict the applicability of prevention mechanisms.

This is also true for API misuses as a prevalent software defect (cf. Section 3.1) [WGMC15, WCH+20]. Therefore, research cannot restrict itself to misuse prevention but has to find solutions for already present API misuses. A necessary step to overcome the negative effect is to understand the root causes of API misuses and design effective techniques for their *detection*.

API misuse research came up with a variety of automated detection techniques, significantly improving API usage. Promising approaches are *pattern-based API misuse detectors*. These detectors infer API usage patterns using Frequent Pattern Mining (FPM) (cf. Section 3.5) based on correct API usage samples. Then, they apply these patterns as specifications to determine violations as misuses. However, currently, these detectors are not applied in practice, particularly due to *low precision* [ANN+19a, ANN+19b] as well as *less practical use cases* (i.e., having assumptions that do not fit into a realistic software development process).

For both issues, namely low precision and less realistic use cases, we suggest two hypotheses for improved and new API misuse detection:

a) Less work has been conducted in improving the input of the FPM of the pattern-based detection. Initial results indicated that filtering the donor code (i.e., code examples as input for mining) of API usages by quality criteria results in patterns achieving larger precision during detection [LW12]. We hypothesize that using information from a currently changed API usage (e.g., obtained via a Version Control System (VCS)) helps to find more related donor code for mining and thus improves the precision of pattern-based misuse detection at this changed code location. [NHO18, NHSO21]

b) Patterns as specifications tend to miss the context in which APIs are applied and thus produce false alarms due to confounding but correct code elements (e.g., an alternative API usage or a workaround). Here, we consider context as *code context*, meaning the surrounding code of the API usage as well as the *context of the change*, namely, whether a similar kind of code was updated in the past to *fix* a misuse. We hypothesize

that these two kinds of *context*, which we represent as so-called *change rules*, improve the precision of API misuse detectors. [NHKO20a, NBKO21a, NBKO22]

Thus, we target the research question **RQ D** of more precise API misuse detection in a realistic software development scenario (i.e., it is, in principle, deployable into a Continuous Integration (CI) process) by analyzing these two hypotheses, which relate to the four following sub-problems:

1. Limitations of State-of-the-art Pattern-based API Misuse Detectors;
2. Applicability of Change-based Information using VCSs for Misuse Detection;
3. Influence of Search and Filter Strategies for Change-based Information on the Precision of Pattern-based API Misuse Detectors;
4. Applicability of Change Rules from previous API Misuse Fixes as a Change Rule-based API Misuse Detection.

Limitations of State-of-the-art Pattern-based API Misuse Detectors. First, we strengthen our hypotheses by analyzing the current *limitations* of state-of-the-art API misuse detectors based on our previous work [NBKO22]. We focused on the two presented hurdles for practical application, namely, the *missing integration in a practical scenario* and the *high false-positive rate* (i.e., falsely reported misuses). We answer this question by reviewing existing literature on API misuse detectors using an SLR-like approach [KC07] based on existing surveys and comparison studies. We concentrated the review on pattern-based detectors since they are the most widely applied group of approaches in API misuse detection research [RBK+13]. Moreover, pattern-based techniques have the advantage of 'learning' from previous usages, and thus allow adaptations of patterns, for instance, if the API evolves due to updates in their library [RBK+13]. Using the results together with the underlying concept of the detector, we present an overview of flaws in their concept that potentially harm the precision as well as practical applicability. Based on this overview, we inferred the restrictions and limitations of these detectors as well as techniques for comparison. Moreover, we obtained a detector to which we compared our techniques.

We summarize this as **RQ D-L** (D-Limitations) as follows:

RQ D-L (D-Limitations): ***What are the limitations of state-of-the-art pattern-based API misuse detectors in practical scenarios?***

We obtained an overview of the state-of-the-art of API misuse detection by reviewing the literature on API misuse detection and identified the limitations with respect to low precision and less practical applicability.

Applicability of Change-based Information using VCSs for Misuse Detection. In the second sub-problem, we target the missing practical integration of pattern-based API misuse detection, particularly their related specification miners. We observed that many pattern-based detectors lack a description of how to obtain the *donor code*, namely the code from which patterns are mined. Moreover, many detectors describe collection processes that are not applicable in practice (i.e., hardly deployable into a software development process). Thus, we used change information of API usages (cf. source code changes in Section 3.4) to further improve the data collection. *Change information (i.e., changed lines of code)*

is available through VCSs, which are de facto standard in modern software development serving as an essential part of CI processes (cf. Section 3.4.1). Thus, we hypothesize that API misuse detection can also be a part of this CI workflow. Based on that, we suggest a concept on how to leverage code changes to obtain information on the API usage in order to find relevant donor code (cf. code search in Section 3.3) for API usage pattern mining. We evaluated this concept by an engineering research approach [RAB+20], particularly by implementing a software artifact named *Relevant API Information Extractor (RAIX)* with subsequent search and filter strategies. We assessed the potential of code changes for finding relevant donor code [NHSO21] and validated whether change-based information obtained by RAIX reduces the search space for potential misuses and whether filter obtained relevant donor code [NHO18, NHSO21].

We summarize this research question as **RQ D-C** (D-Change-Information) as follows:

RQ D-C (D-Change-Information): ***Is change information a meaningful source for finding related API usage samples for API misuse detection in practical use cases?***

We evaluated whether a concept of leveraging change information from VCS can be applied in realistic software development by using the engineering research approach with an own software artifact. We found that change information significantly reduces the number of code locations for API misuse detection analysis.

Influence of Search and Filter Strategies for Change-based Information on the Precision of Pattern-based API Misuse Detectors. In the third sub-problem, we analyze whether filtering change-based information that we collect before eventually improves the precision of API misuse detection. In detail, we assume a client developer (cf. Chapter 1) who changes their code and introduces an API misuse. Due to the change information, we know how and which API elements are changed. By using this *change information for searching and filtering donor code* for API usage pattern mining, we expect to find *more related patterns* regarding the changed API usage in the client code. This way, we hypothesize an *improvement in the precision of the subsequent pattern-based API misuse detection* since the patterns inferred and applied for detection exhibit higher relevance to the changed API usage. We suggest different filtering scenarios. Once again, we evaluated these concepts using engineering research [RAB+20] by a software artifact. Consequently, we give evidence for research in pattern-based API misuse detection on whether and to what degree change-based information improves the relevance of patterns for detection and, subsequently, the detectors' precision [NHSO21]. The results also provide a practical scenario as well as insights into the applicability of state-of-the-art API misuse detectors leveraging the VCS.

This is summarized as **RQ D-F** (D-Filtering) as follows:

RQ D-F (D-Filtering): ***What is the impact of the previous filtering of the donor code from which API patterns are mined on the subsequent pattern-based API misuse detection?***

We evaluated the effect of different variants of change-information-based filtering strategies for the donor code of FPM of API specifications on the resulting relevance of patterns for detection and their impact on the precision of pattern-based API misuse detectors by conducting engineering research with a software artifact. We identified a set of search and filter mechanisms (e.g., internal filtering, filtering with knowing the misused API), based on whichAPI misuse detection improved.

Applicability of Change Rules from previous API Misuse Fixes as a Change rule-based API Misuse Detection. Finally, we target the second hypothesized limitation of *missing relevant code and change context*. If missing context, a detector would apply non-relevant patterns as specifications, causing potential false positives. Our hypothesis is that using this contextual knowledge improves the precision of the misuse detection. Particularly, we suggest a concept in which previous fixes of API misuses are used to extract their essential edits as change rules (i.e., using change context). We refer to this concept as *Change Rule Inference (ChaRLI)* [NHKO20a, NBKO21a, NBKO22]. This notion aligns with research using historical knowledge of automated API migration [MWZM12, LS18] or general Automated Program Repair (APR) [LLLG16, HW18]. However, these have – to the best of our knowledge – not been used for API misuses in particular and thus did not tackle their specific characteristics. We leveraged these characteristics by requiring a code representation (cf. Section 3.2), particularly, we applied a graph representation of an API usage introduced by Amann et al. [ANN+19b] presented in Section 3.2.2. Moreover, we tested different variants of change rules containing more surrounding code elements (i.e., using code context). Our concept applies the characteristics of API misuses (e.g., referred data type), and thus, we expect a beneficial effect precision on API misuse detection. We refer to it as *change rule-based API misuse detection* based on a technique named *Change Rule-based API Misuse Detection (RuDetect)* [NHKO20a, NBKO21a, NBKO22]. Once again, we used engineering research [RAB+20] with a software artifact to evaluate our concept.

The research question is summarized as **RQ D-R** (D-Rules) as follows:

RQ D-R (D-Rules): ***What is the impact of applying change rules inferred from previous fixes of API misuses on API misuse detection?***

We evaluated whether leveraging previous change rules from prior API misuse fixes improves the precision of API misuse detection. We analyzed this improvement by a software artifact of a rule-based API misuse detector. We identified a positive significant impact on the precision compared to a state-of-the-art API misuse detector.

Structure of RQ D In Figure 2.2, we depict the structure of sub-research questions of **RQ D**. In detail, the limitations (i.e., **RQ D-L**) are the basis for the three sub-questions on the applicability of change-information (i.e., **RQ D-C**), filtering strategies (i.e., **RQ D-F**), and change rule-based misuse detection (i.e., **RQ D-R**). Moreover, the applicability of **RQ D-C** is a mandatory step for **RQ D-F**, since otherwise, the improved pattern-based misuse detection would lack its practical use case.

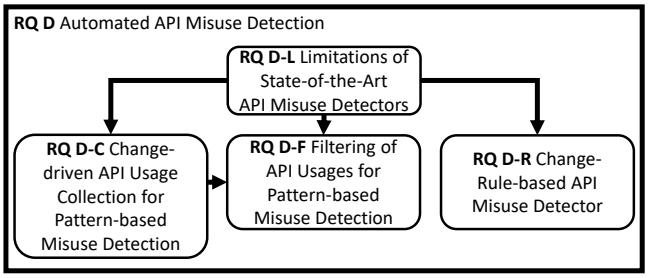
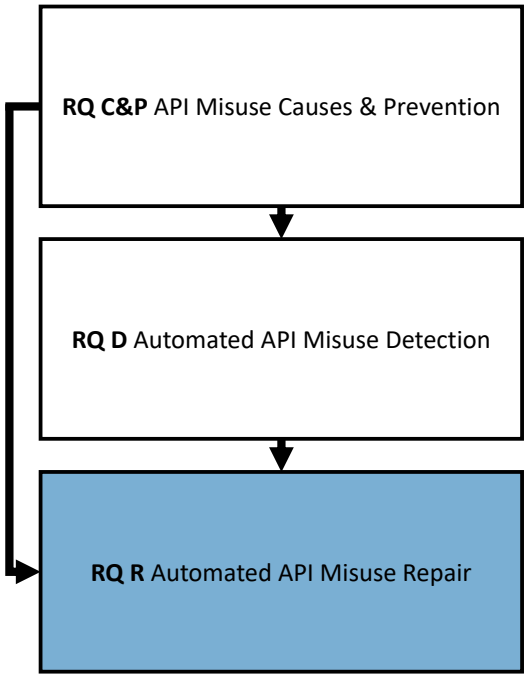


Figure 2.2.: Structure of the Sub-Research Questions of **RQ D**

2.3. RQ R Automated API Misuse Repair



Research on API misuses has a focus on their automated detection. While this is generally beneficial, fixing misuses is still a manual process. Nevertheless, the domain of APR has made tremendous advances in the last decade of research [Mon18a, LGPR19] as well as in practical applications [NHL18, BSPC19, KWM+21]. Nevertheless, current APR techniques do not consider API misuses in particular. As discussed previously, no general solution for APR exists, and thus, specializing in single APR techniques for specific defects, such as API misuses, is reasonable [NL22].

Particularly, we target the research question **RQ R** by considering whether an API-specific APR technique leveraging data structures from the previous misuse detection (i.e., either pattern- or change rule-based) produces useful code patches. We do this by analyzing three sub-problems, namely,

1. Limitations of the state-of-the-art APR techniques regarding fixing API misuses;
2. Applicability of API usage patterns from pattern-based misuse detection for APR;
3. Applicability of change rules from change rule-based misuse detection for APR.

Limitations of the State-of-the-art APR Techniques Regarding Fixing API Misuses. According to the first sub-problem, we reviewed the literature on APR research. Particularly, we used the existing survey by Monperrus [Mon18a] and assessed the ability of state-of-the-art APR techniques to target API misuses. We also analyzed the publication by Kechagia et al. [KMSH21], who had evaluated APR techniques in particular for API

misuses. This way, we pointed out essential *limitations* of APR techniques with regard to API misuses and thus extend our previous work [Nie17].

We summarize this as **RQ R-L** (R-Limitations) denoted as follows:

RQ R-L (R-Limitations): ***What are the limitations of state-of-the-art APR techniques to repair API misuses?***

We assessed the limitations of state-of-the-art APR techniques for API misuses based on previous literature reviews on APR and its limitations, as well as contrasted their abilities regarding the special characteristics of API misuse as opposed to generic defects. We found that APR techniques are limited regarding test dependencies, benchmarks, and insufficient patches specific to APImisuses.

Applicability of API Usage Patterns from Pattern-based Misuse Detection for APR. In **RQ R-P**, we leveraged patterns used for misuse detectors as possible repair templates. The hypothesis is that patterns representing a specification of correct behavior can be used to transform the actual misuse into the correct usage represented by the pattern. By applying an engineering research methodology [RAB+20], we developed a software artifact named *API-Specific Automated Program Repair (ASAP-Repair)*, which can repair misuses in the form of an intermediate code representation of API Usage Graphs (AUGs) introduced in Section 3.2.2. ASAP-Repair provides - to the best of our knowledge - *the first step towards API-specific APR*, which we refer to as *pattern-based repair* variant. We evaluate the patches produced by the pattern-based repair of ASAP-Repair using real-world API misuse datasets.

RQ R-P (R-Patterns): ***Do API usage patterns provide benefits for API-specific APR?***

We suggested a concept of applying patterns from pattern-based API misuse detection as part of a pattern-based APR technique within our software artifact ASAP-Repair and assess its repair ability (i.e., by generating patches in the intermediate representation AUGs) for real-world API misuses. We found that pattern-basedASAP-Repair can repair in an idealized but not necessarily in a realistic scenario.

Applicability of Change Rules from Change Rule-based Misuse Detection for APR. In the last sub-research question, **RQ R-R**, we analyzed whether change rules from our improved change-based misuse detector (i.e., **RQ D-R**) also improve the repair ability compared to standard APR. Conceptually, we hypothesized that change information from previous API misuse fixes can be leveraged to produce patches for API misuses. This notion is reasonable as, for instance, general APR techniques also apply historical information to fix bugs [LLLG16, HW18]. Again, we applied the methodology of engineering research [RAB+20], developed a concept, and implemented it in the same software artifact ASAP-Repair, which uses change rules to produce a patch in the form of the intermediate representation of an AUG. We referred to this technique as *change rule-based API misuse repair* variant. We evaluated its repair ability and thus compared it to the pattern-based repair variant from **RQ R-P**.

We summarize this research question as follows:

RQ R-R (R-Rules): ***Do change rules of previous API misuse fixes provide benefits for API-specific APR?***

We developed a concept using change rules from previous change rule-based API misuse detection as part of the change rule-based repair in our software artifact ASAP-Repair. We assessed its repair ability (i.e., by generating patches in the intermediate representation AUGs) for real-world API misuses and in comparison to pattern-based repair. We found that ASAP-Repair can repair API misuses in an idealized and in a realistic scenario.

Structure of RQ R In Figure 2.3, we depict the structure of sub-research questions of **RQ R**. In detail, the limitations of current APR techniques (i.e., **RQ R-L**) form the basis for analyzing the ability of *pattern-based* (i.e., **RQ R-P**) and *change rule-based* repair (i.e., **RQ R-R**)

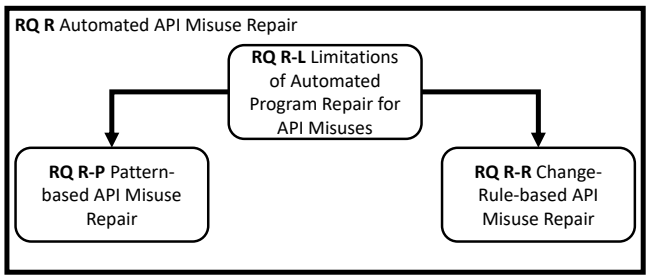


Figure 2.3.: Structure of the Sub-Research Questions of **RQ R**

2.4. Detailed Structure of the Research Questions in Thesis

In Figure 2.4, we present the detailed structure of the thesisin the chapters 4, 5, 6, and 7, which have a special focus on answering the three main research questions **RQ C&P**, **RQ D**, and **RQ R**. This overview serves as a shortcut to find the chapters concerned with the respective research question.

Moreover, it depicts the relations of the research questions. In detail, **RQ C&P** serves as the necessary background on why client developers misuse APIs, which helps to assess the applicability of detection (i.e., discussed in **RQ D**) and APR techniques (i.e., discussed in **RQ R**).

As mentioned before, a necessary step of APR is the detection of a code defect, and thus, API misuse detection is a mandatory component of API-specific APR. On the one hand, limitations of pattern-based misuse detectors (i.e., **RQ D-L**) influence pattern-based repair (i.e., **RQ R-P**). On the other hand, having an improved technique incorporating change information into API specification mining (i.e., **RQ D-C**) and subsequent filtering resulting in improved pattern-based detection (i.e., **RQ D-F**), has a beneficial impact on pattern-based repair.

Finally, the change rule-based repair technique (i.e., **RQ R-R**) requires change rules and the inference technique ChaRLI (i.e., discussed in **RQ D-R**).

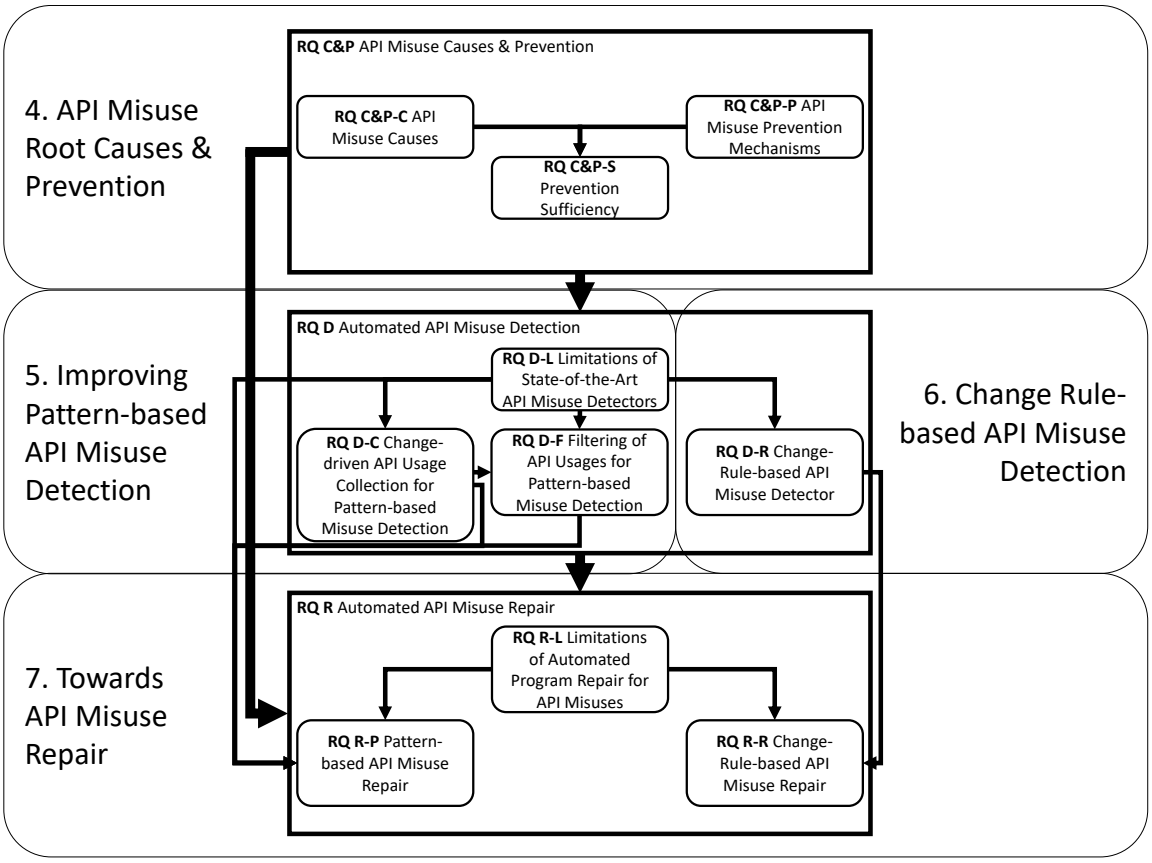


Figure 2.4.: Detailed Structure of the Research Questions.

3

Fundamentals and Background

This chapter describes basic concepts and body of knowledge on related topics to understand the techniques and design decisions within this thesis, especially in the main chapters 4, 5, 6, and 7. Section 3.1, we provide an overview of studies of *software defects* with a focus on their prevalence, severity as well as generic software defect detection techniques, and particularly, a specific consideration of *Application Programming Interface (API) misuses*. Then, we introduce basic *code representation* concepts as well as *specific code representations of API usage* in Section 3.2. In Section 3.3 introduces concepts of *finding related sources*, namely the topic *code search*. Subsequently, we present basics of *source code changes* and related problems and possibilities regarding API misuses (cf. Section 3.4). Finally, we present the *concept of Frequent Pattern Mining (FPM)* (cf. Section 3.5), a technique applied within prior API misuse detection.

3.1. Software Defects and API Misuses

In this section, we discuss the background of *software defects*, also frequently denoted as *bugs*, and thus, we apply both terms interchangeably. In detail, we consider related studies on bugs (cf. Section 3.1.1) and applied techniques for bug detection (cf. Section 3.1.2). Afterward, we consider related work on the taxonomy and prevalence of API misuses, the kind of bug that we investigate in this thesis.

3.1.1. Studies of Software Defects

We discuss related studies on bugs regarding their considered *software defect types*, the *data used for analysis*, as well as which *aspects of bugs* are analyzed.

Software Defect Types Widder and Le Goues [WG24] approached the topic of *what denotes a bug* from a philosophic perspective. In detail, they emphasized that while initially, the answer to this question seems straightforward, on a second view, things can become much more blurred. For instance, they ask for specific conditions to decide whether a certain behavior denotes a bug. Do we say something is a bug if it is reported by a user in an issue tracker, if a sufficiently large user base complains about this behavior, or if the developer finally accepts it in an issue tracker? Especially bugs concerning the look and feel or which harm a small user base (e.g., blind persons) might fail the last condition due to a `Won't fix` resolution.

Based on this consideration, it is not surprising that a generally accepted and complete overview of software defects does not exist and maybe never will. Therefore, empirical

studies on software defects can only take specific perspectives and focuses limited regarding their interpretation of the term 'bug' as well as their bug selection criteria.

Zhong and Su [ZS15] focused on bugs obtained via patch fixes and, thus, bugs that were *found to be worth fixing in the past*. Other studies had a focus on *specific developers*, for instance, beginners [BCHH10]. Another set of studies explored bugs with a *specific temporal or inter-bug characteristic*, such as long-remaining bugs [CNSH14, SKP14] or critical blocking bugs [RLC20] (i.e., bugs blocking the fix of other bugs). Bug studies also analyzed certain *specific bug types* based on their occurrences in the code, such as regular expression bugs [WBJS20], exception handling bugs [NVN19], or test code bugs [VFM15, TSM18], as well as their specific bug behavior, for instance, concurrency bugs [BFSK20] or performance bugs [HY16]. Finally, several studies focused on certain *application domains*, such as Deep Learning (DL) [ZCC+18, INPR19, JZW+20, YHXF22], quantum machine learning [ZWL+23], mobile applications [XXS+23], platform management systems (e.g., Kubernetes) [XGW24], or the evolution of bugs in open-source software [LTW+06].

Analyzed Data Empirical studies obtained bugs from various sources. Most frequently, researchers used *issue tracking systems* of projects, in which users and developers report and manage bugs of a software [LTW+06, HY16, XXS+23, ZWL+23, XGW24] as well as *Version Control Systems (VCSs)* [LP17, TSM18, ZCC+18, BFSK20, WBJS20] (e.g., based on commit messages or pull requests) as well as a *combination of both sources* [CNSH14, VFM15, ZS15, INPR19, NVN19, JZW+20, RLC20, YHXF22]. Other classical sources are *Q&A pages* (e.g., StackOverflow[1]) [ZCC+18, BFSK20], *existing bug databases* [SKP14, LP17], or *developer experiments* [BCHH10].

Analyzed Aspects Studies derived *taxonomies of bugs* [BCHH10, CNSH14, INPR19], partially enhanced with a consideration of their *prevalence* [VFM15, HY16, INPR19, NVN19, YHXF22] as well as the *evolution of the prevalence* [LTW+06]. This way, researchers can focus on the most frequently occurring bugs. Based on the assessed severity of bugs, as studied by Saha et al. [SKP14], the aspect of importance is considered as well.

Another frequently analyzed aspect is the *root causes* of different bug types [LTW+06, CNSH14, VFM15, HY16, TSM18, ZCC+18, INPR19, NVN19, BFSK20, JZW+20, RLC20, WBJS20, YHXF22, XXS+23, ZWL+23, XGW24]. These considerations help to develop effective prevention mechanisms.

Moreover, the concrete *symptoms or manifestations, as well as implications of bugs*, are part of many studies [LTW+06, SKP14, HY16, ZCC+18, NVN19, BFSK20, JZW+20, WBJS20, YHXF22, XXS+23, ZWL+23, XGW24]. Based on this knowledge, researchers and practitioners can assess the consequences of certain bug types and plan mitigation strategies.

A further aspect is the *fix types and fix characteristics of bugs* [CNSH14, SKP14, VFM15, ZS15, HY16, LP17, TSM18, NVN19, RLC20, WBJS20, YHXF22, XGW24]. Both enable developers to derive fix patterns or researchers to develop Automated Program Repair (APR) techniques [LGPR19].

Further, researchers investigated *bug locations* [ZS15, JZW+20], *related process steps* [INPR19], and *detection and location techniques* [ZCC+18, XXS+23].

[1] https://stackoverflow.com/ last accessed: 2025/04/15

3.1.2. Software Defect Detection

We denote *software defect detection* as the recognition that a software defect or bug is present (e.g., an accepted issue in an issue tracker) using techniques and processes for their reproduction [Zel06, p. 85pp] and location [WGL+16]. Note that this is different from techniques assessing software defect-proneness, which determine the probability of a software component containing a bug without providing a real manifestation of an instance of it [SJS+11].

Defect detection should preferably occur at the development stage of the software since end-users of the software would not be bothered by crashes or unexpected behavior of the software in production [Som18, p. 256pp]. Nevertheless, in case end-users or developers recognize a potential software defect, it is standard to report these in an issue tracking system [Zel06, p. 27pp].

Manual Defect Detection Regarding bug detection in the development phase, there exist several manual processes such as code walk-throughs (i.e., developers present their implemented solution to other developers), code reviews (i.e., assessing the implemented solution by other developers), or formalized code inspections (i.e., formal individual and group checks of the implemented code) [RAT+06] [Wag13, p. 121pp]. While formal processes have been shown effective [Wag13, p. 124p], they are typically very costly and thus merely worth it for severe bugs and critical systems. Therefore, the lightweight process of *modern code reviews* is more applicable in practice, integrating code review in a Continuous Integration (CI) process, for instance, via the pull request mechanism by GitHub[2] [BBZJ14]. Note that reviews or third-party audits of software (e.g., applied in a certification process) not only encompass software product quality by the code itself but also consider its requirements, its software architecture, and its design, as well as an assessment of the software development process [WTH+24, p. 1-15p, p. 2-10p, p.3-13p, p. 12-1pp].

Automated Defect Detection Additionally, research effort was put into *automated bug detection*. A typical distinction is between *static* and *dynamic techniques* where static techniques consider the source code and binaries without executing them [APH+08], while dynamic techniques run the program, for instance, with test data [WTH+24, p. 5-2].

Prominently applied *static approaches* are static code analysis [JSMHB13, ML23] [Wag13, p. 128pp]. These comprise simple style checkers such as Checkstyle[3], bug pattern detectors such as PMD[4] and FindBugs[5] [APH+08] up to data and control analyzers such as Soot[6] or its successor SootUp[7] [KSK+24] as well as formal verification [Som18, p. 337pp] and analysis techniques [WTH+24, p. 12-13] with tools like Java Pathfinder[8].

The most prominently applied *dynamic technique* is automated testing [RAT+06], [Zel06, p. 53pp] [Wag13, p. 134pp] [WTH+24, p. 5-1pp], which comprises steps to design test cases, to create test input data, to execute test cases as well as to track and compare test

[2] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/requesting-a-pull-request-review last accessed: 2025/04/23
[3] https://checkstyle.sourceforge.io/ last accessed: 2025/04/23
[4] https://pmd.github.io/ last accessed: 2025/04/23
[5] https://findbugs.sourceforge.net/ last accessed: 2025/04/23
[6] https://soot-oss.github.io/soot/ last accessed: 2025/04/23
[7] https://soot-oss.github.io/SootUp last accessed: 2025/04/23
[8] https://github.com/javapathfinder last accessed: 2025/04/23

results [Som18, p. 260]. Automated testing is also accompanied by debugging, program slicing, and logging techniques to essentially locate the bug origin [Zel06, p. 145pp, p. 167pp, p. 199pp]. Moreover, to reduce the manual effort of writing test cases, techniques for automated test case generation [FA11, SJR+15] up to fuzzing, which aim to generate a large set of test cases and input data to trigger hidden and sparse bugs (e.g., security issues) [ZWCX22], were developed.

3.1.3. Taxonomy and Prevalence of API Misuses

We further consider the special software defect of API misuse, which we already defined as a software defect characterized by a *deviant API usage apart from the intended usage by the API developer, which causes a negative software behavior*. In detail, we present different *taxonomies* and the *prevalence* of API misuses in light of ordinary software defects, as these further emphasize the characteristics and relevance of API misuses.

API Misuse Taxonomies There exist a few classifications of API misuses. Amann et al. [ANN+19a] introduced the "API-**Mis**use **C**lassification (**MuC**)" based on their previously collected API misuse dataset *MUBench* [ANN+16]. They defined four API elements than can be misused, namely,

- *method calls*;
- *conditions*, which they further detailed into checks for `null` values, certain values or states, synchronization mechanisms, or a certain context;
- *iterations*;
- and *exception handling*.

For all elements, they found examples of *missing* and *redundant* API elements in MUBench.

Li et al. [LJB+21] extended this classification based on an analysis of 528,546 bug-fix commits from GitHub and added a further violation type of *replaced* API elements. These denote false elements, which have to be exchanged for the correct ones. In detail, they discussed replaced parameters, names, and receiver objects of method calls.

Based on a literature review, Schlichtig et al. [SSNB22] harmonized the "**F**ramework for API **U**sage constraint and **M**isuse classification (**FUM**)." First, they defined *API misuses* as violating *API usage constraints*. They denoted API usage constraints as a subclass of *API directives* that restrict a certain API usage (e.g., enforcing certain values for parameters), and thus, a violation causes a misbehavior, such as crashes or vulnerabilities. They referred to the more general term of API directives as a description of the correct and optimal usage of an API (e.g., as statement in the documentation). Since they bind API misuses to violated API usage constraints, they classify misuses based on the usage constraints. In detail, they extended a taxonomy by Monperrus et al. [METM12] on API directives and declared constraints related to three parts of an API method, namely, its *return value*, its *method call*, and its *parameters*. Then, they distinguish different API misuses based on constraints for:

- the *return value*, namely, their constrained post-calls and post-null-checks;
- the *method call*, namely, their constrained call sequence, control of the method call (e.g., using conditions), and forbidden method calls (e.g., outdated methods);

- the *parameters*, namely, their constrained state, string format, number range, pre-null-check, method parameter type, and method parameter correlations;

- *multiple parts*, namely, constrained exception handling, context (i.e., threading and synchronization contexts), and other high-level constraints going beyond the API method scope (e.g., interactions with the operating system).

Prevalence of API Misuse In general, a few studies analyzed the amount of API misuses among ordinary software defects. In detail, those studies considered historic bug fixes and decided whether these fixes denote an API misuse. Zhong and Su [ZS15] found that half of the changed source files within bug fixes in five open-source projects required at least one API-related change. A similar amount of 51.7% of edit operations related to API misuses was observed by Li et al. [LJB+21] by analyzing 528,546 bug-fix commits from 220,053 projects from GitHub. Gu et al. [GWL+19] determined that ≈ 17% of bug fix-related commits from six open-source projects are related to API misuses. Thus, we denote that API misuse represents a prevalent software defect type.

Using their introduced taxonomy MuC, Amann et al. [ANN+19a] found in the same study using their previously collected dataset MUBench [ANN+16] that more missing than redundant API elements exist (i.e., 89 vs. 21 misuses) as well as most misuses were classified as conditions (i.e., 52) and method calls (i.e., 43). Similar results were obtained by Kang and Lo [KL21] on their AU500 dataset consisting of 115 API misuses, which only consists of missing API elements and most frequently conditions (i.e., 62) and method calls (i.e., 51). Using the extension of MuC, Li et al. [LJB+21] classified API edit operations based on 528,546 commits and also found more missing than redundant API elements (219,338 vs. 64,913). Most frequently, they found replaced API elements (i.e., 220,969) followed by API calls (i.e., 196,307)[9]. Gu et al. [GWL+19] did not refer to a known classification, however, found based on a sample of 830 API misuses that the most frequent one denotes violations of the order of method calls (i.e., 27.2 − 42.5%), followed by improper error handling (i.e., 19.5 − 34.1%), and improper parameter usage (i.e., 14.3 − 19.5%).

3.2. Code Representation for Code Analysis

In this section, we briefly discuss general-purpose code representations (cf. Section 3.2.1) and a more detailed discussion on API-specific code representation (cf. Section 3.2.2), most importantly on API Usage Graphs (AUGs), a data structure subsequently applied in Chapters 5, 6, and 7.

3.2.1. General Code Representations

While source code representation for human developers is typically in textual form, compilers, auxiliary tools in a development environment, and code analysis tools usually rely on dedicated code representation. Most prominently known are token streams [ALSU14, p. 43pp], Abstract Syntax Trees (ASTs) [Som18, p. 217pp], or just syntax or parse

[9] Li et al. [LJB+21] reported percentages for the related API misuse classes in Table 1 in their publication. However, we were not able verify these numbers since the absolute reported number of API operations (i.e., 576,515) differs from the sum of all distinct API operations from that table (i.e., 505,220).

trees [ALSU14, p. 45pp]. Moreover, Control Flow Graphs (CFGs) [All70], Program Dependence Graphs (PDGs) [OO84] were invented to support intraprocedural code representation (i.e., within a method declaration), while system dependence graphs [HRB90] allow interprocedural code representation (i.e., among method declarations).

3.2.2. API-Specific Code Representations

Even though general-purpose code representations are applicable to represent an API usage, such as demonstrated with ASTs [LZ05] or CFGs [JWL+24], research elicited code representation particularly tailored for API misuses, namely, Graph-based Object Usage Models (Groums), API Usage Graphs (AUGs), and extended AUGs.

Graph-Based Object Usage Model Nguyen et al. [NNP+09b] introduced the data structure of the Groum. It denotes a directed acyclic graph consisting of action nodes, representing invocations (e.g., method calls, field accesses), and control nodes, representing branching points in the code structure (e.g., `if`-conditions or loops). A Groum represents the usage of multiple objects within a method declaration scope. The nodes are connected by non-labeled edges representing the order and data dependencies. This way, the authors emphasize that Groums denote a more compact representation than CFGs and PDGs. Groums are automatically built using the ASTs within their GrouMiner tool.

API Usage Graph Amann et al. identified that current API misuse detectors suffer from an imprecise API usage representation, causing a significant amount of false negatives (i.e., not detecting misuses on average 45.8% in their analysis of related API misuse detectors) [Ama18, ANN+19a]. Thus, they suggested a new graph-based data structure, API Usage Graph (AUG), based on the previously described Groums. Similar to a Groum, an AUG contains control as well as data flow information statically obtained from the AST. The AUG concentrates on the representation of intraprocedural API usages (i.e., API usages within an individual method declaration). Amann et al. achieved promising results by applying AUGs for API misuse detection [Ama18, ANN+19b], and thus, we used and extended this data structure based on their publicly available replication package[10]. Note that there exists no formal specification on how to transform an AST to an AUG, and thus, the implementation mainly forms a loose definition. Moreover, the definition of an AUG mainly relies on the Java programming language.

Due to the prevalent usage of AUGs in this work, we present the main characteristics of an AUG based on a small example presented in Figure 3.1, together with a graphical representation also suggested by Amann et al. [ANN+19b]. Nevertheless, we also present a formal definition of an AUG because first, this denotes a more precise and unambiguous presentation of our interpretation of an AUG, and second, we use this definition to define further concepts, such as similarity metrics introduced in Section 6.3.5.

[10] available within their GitHub repository of their MUDetect tool `https://github.com/stg-tud/MUDetect` last accessed: 2023/06/30

Formally, we define an AUG *aug* as a directed, labeled, acyclic multigraph

$$aug := (V, E, \Sigma_V, \Sigma_E, s, t, l_V, l_E)$$

where

- V denotes a set of nodes,
- $E : V \times V$ is a multiset of edges,
- Σ_V and Σ_E denote finite alphabets of labels for nodes (i.e., V) and edges (i.e., E),
- $s : E \to V$ and $t : E \to V$ define functions mapping an edge either to its source node (i.e., s) or its target node (i.e., t),
- $l_V : V \to \Sigma_V$ and $l_E : E \to \Sigma_E$ denote the labeling functions.

From our sample code (cf. Code Listing 3.1), we extracted the AUG representing the `doSomething` method and represented it in its graphical form (cf. Figure 3.1). In general, the AUG consists of two main kinds of nodes, named *action* nodes (i.e., rectangles in Figure 3.1) and *data* nodes (i.e., ellipses in Figure 3.1). An *action node* can represent an API method call (e.g., node `QClass.requiresMore()`) or certain control blocks (e.g., node `<return>` denoting the `return` statement). Depending on the specific sub-type, action nodes are labeled with their particular method name or control block (i.e., denoted by the angle brackets `<...>`). *Data nodes* instead represent object instances and variables labeled by their type (e.g., node `BClass`) or constants that are labeled by their raw value (e.g., node `42`). The type resolution is made statically based on the currently parsed AST as well as the enclosed source files and libraries[11]. In case certain source code is missing, or the type resolution fails to infer any type, a data node is labeled as `UNKNOWN`. In its current implementation[12], an AUG can consist of 19 different action node types and six different data node types. Formally, we determine the specific type of a node by defining the function $type : V \to String$, which returns the type name as *String*-value. As an extension to the AUG, we also define the function $api : V \to String$, which returns for each node in the AUG its corresponding fully qualified type name (e.g., `java.lang.Object` for node `Object.<init>`) if it is resolvable.

Furthermore, the nodes in an AUG are connected via *data flow* (i.e., solid edges in Figure 3.1) and *control flow* edges (i.e., dashed edges in Figure 3.1). In general, control flow edges connect pairs of action nodes, while data edges connect pairs of data and action nodes. In our example, we can observe that data flow edges can represent

- the usage of a value as a parameter in a method (i.e., `para`, such as data node `42` for action node `Object.<init>`),
- the usage as a result of a method call (i.e., definition edge labeled as `def`, such as action node `Object.<init>` that produces the object of type `QClass`),
- the usage as receiver object, on which an action is performed (i.e., `recv`, such as data node `QClass` on which the action `QClass.requiresMore` is performed).

[11] internally, MUDetect uses the JDT type resolution from Eclipse `https://eclipse.dev/jdt/` last accessed on 2023/07/05

[12] date on the time of writing on 2023/06/30

```
1  package my.own.pkg.subpkg;
2
3  import a.b.AClass;
4  import a.b.BClass;
5  import a.b.CClass;
6  import x.y.ZClass;
7  import my.own.pkg.QClass;
8
9  public class Foo extends AClass {
10
11  BClass myB = new BClass(1337,"Bar");
12  CClass myC = new CClass();
13
14  protected ZClass doSomething(BClass bObj) {
15   System.out.println("do Something")
16   QClass myQObj = new QClass(myB, 42);
17   if(myQObj.requiresMore()){
18    myQObj.addData(myC);
19   }
20   return myQObj.merge(bObj);
21  }
22 }
```

Code Listing 3.1: Fictive code sample of an API usage depicted as AUG in Figure 3.1.

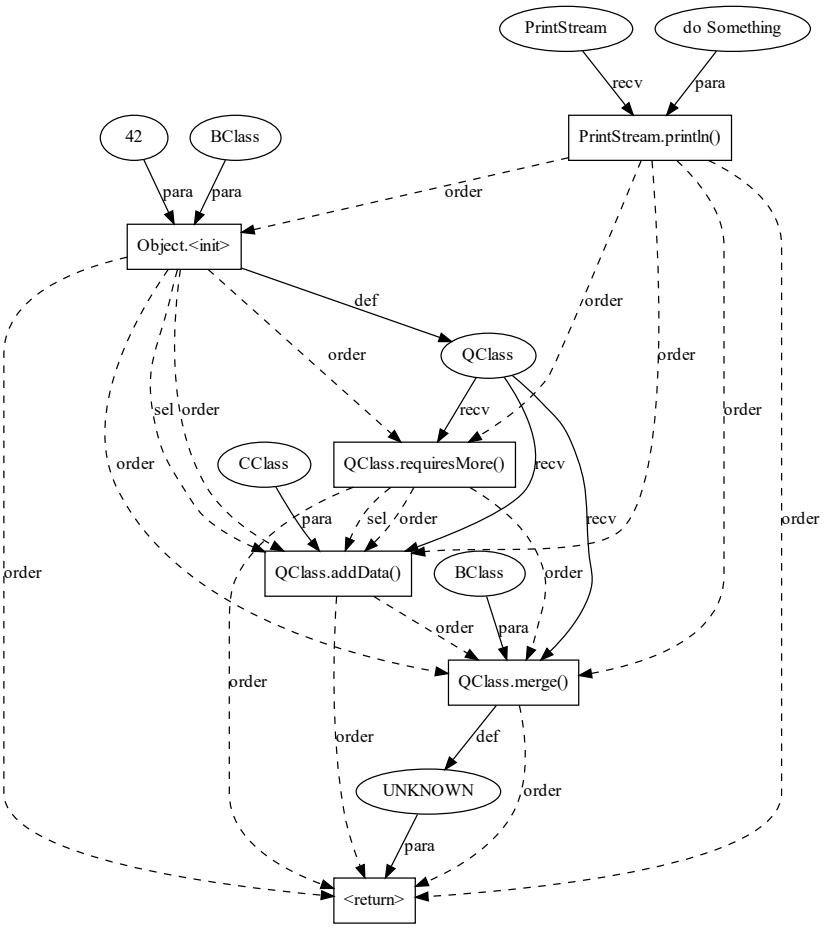


Figure 3.1.: Graphical representation of the corresponding AUG in Code Listing 3.1.

In our AUG sample, the control flow edges describe

- the usage to denote the order between action nodes (i.e., `order`-edges, such as the `Object.<init>` action is performed before the `QClass.requiresMore()` action)
- the usage to denote a branching of action as indicated by `if`-conditions (i.e., selection edge labeled as `sel`, such as the action `QClass.requiresMore()` condition and the action `QClass.addData()`)

Note that due to static inference of the control and data flow information from an AST, its precision is limited. Therefore, for instance, `order`-edges are generated conservatively by forming a transitive closure between all pairs of subsequent action nodes. This transitive closure drastically increases the number of `order` edges when having a larger number of action nodes. Moreover, as indicated by the example, an AUG – in opposite to its predecessor, the Groum – represents control structures as edges instead of nodes, such as `if` conditions (i.e., selection edges labeled as `sel`) or loops, namely, `for`, `while`, or `do ... while` constructs, (i.e., repetition edges labeled as `rep`). While Amann et al. gave no clear justification for why they applied edges instead of nodes, it can be assumed that they intended to reduce the overall number of nodes to keep an AUG as compact as possible. The current implementation has four different sub-types of data flow edges and eight different control flow edge types, which label the respective edges. Even though an AUG is a multigraph, it allows at most one edge per type between a pair of nodes. Therefore, the type of the edge can be formally obtained by the labeling function l_E, and thus, $|\Sigma_E|$ denotes the number of different edge types.

Extended API Usage Graph Kang and Lo [KL21] suggested an extension to AUGs named EAUG to better depict certain characteristics of API usage. Similar to the original AUG, these descriptions are based on the Java programming language and thus may be restricted to these language specifics. The extensions mainly refer to more inserted edges as well as additional node and edge types. Particularly, they

- handle different variables originating either from fields (i.e., from the class declaration) and parameters (i.e., from the method declaration);
- take the different execution order of initialization (e.g., constructors, field initializers) into account;
- explicitly mark subtyping and inheritance, which provides the information on usages of own APIs or implementation of interfaces.

Based on their extension, they observed an improvement when applying their misuse detection approach **ALP** (i.e., **A**ctively **L**earned **P**atterns) presented in Section 5.2.4.

3.3. Finding Relevant Source Code Samples

A central idea of API misuse detection is to infer frequent patterns from previous code examples. Therefore, a crucial element is to *find relevant code examples* from which patterns can be inferred. This related software engineering task is known as *code search*. Thus, we consider general aspects of code search (cf. Section 3.3.1) as well as specific code search related to APIs (cf. Section 3.3.2) in this section.

3.3.1. General Code Search

From previous surveys [LXL+21, DGP23], we know that code search is *a software engineering task* focused on finding *relevant code examples* from a *code base*. Relevance is related to the *developer's intent expressed by search queries* (e.g., as textual description or already written source code).

The most frequently applied use case is code reuse (i.e., it represents 90% of the search effort according to Liu et al. [LXL+21]). Nevertheless, retrieved code examples also serve various other tasks such as code comprehension, learning APIs, bug and vulnerability detection, code repair, and code synthesis [GVES09, SSE15, XBL+17, LXL+21, DGP23, CPS24].

The research effort targets different aspects of code search. One aspect denotes the applied query types, representing the search intent of the developer, which ranges from informal text queries, code snippets (e.g., single methods and types, input-output examples, or test cases), and queries written in domain-specific languages up to formal specifications [PP94, HM05, SC06, MGP+11, SED14, LXL+21, DGP23].

Other researchers concentrate on the indexing mechanisms, which – similar to information retrieval techniques – structure the code base (e.g., targeting the source code, binaries, or runtime trace indexing) to obtain faster and more effective code search [GVES09, LXL+21, DGP23].

Other aspects refer to the granularity (e.g., single code elements, methods, classes, applications, or libraries) and the ranking mechanisms of search results (e.g., distance and similarity metrics as well as clustering and filtering techniques) [LXL+21, DGP23].

The research area of code search assesses the recent advances in machine learning as very impactful, particularly on Large Language Model (LLM). In detail, such models can improve the comprehension of complex queries, the comparison mechanisms for code and queries, the ranking mechanisms, the code clone search, the personalized search results, or the answers to questions beyond code search (e.g., generating test code) [GZK18, LXL+21, DGP23, CPS24]. Therefore, applying machine learning is an active domain. Another open challenge in code search is the detection and representation of the user's intent [CPS24].

3.3.2. API Code Search

There exist different flavors within API-specific code search. One branch considers the *recommendation of entire libraries* for certain use cases [TLL13, WHW+22, LZP+23]. Another approach *suggests potential features within a library* [TWLL13]

Most frequently, we observe API code search targeting client developers by *suggesting API elements* (e.g., single methods) *up to API code examples or API usage patterns*. Typically, these techniques use a *known API type* together with other code contexts or queries to retrieve relevant API usages. For instance, `Prospector` [MXBK05] suggests an object creation process by finding code snippets taking existing parameter types together with a targeted output type. Saul et al. [SFDB07] proposed a technique to find similar API methods given an API type and a certain function. There exist techniques applying textual queries [CCL12], partially extending the code base to search in with the API documentation [LZL+15], and Q&A pages [RRL16] to improve search results. `MUSE` [MBDP+15] and `AUSearch` [ATLJ20] suggest real-world code examples for single API methods and types. With the trend of machine learning techniques, specific API embeddings were developed, which allow cross-language API code search [NNPN17, HXX+18, CXLX21a].

3.4. Source Code Changes

In our work, we leverage change information of source code to detect and repair API misuse. These techniques are grounded on automated tools support of VCS presented in Section 3.4.1, as well as the technique of repository mining, which we present in Section 3.4.2. Moreover, we also discuss the general background of the research on API evolution, which shares similarities to our discussed techniques as well as representing a potential cause for API misuses (cf. Section 3.4.3).

3.4.1. Version Control Systems

According to Sommerville [Som18, p. 864], *version control* refers to a process that enables developers to *trace and manage changes in a software product*, particularly to *restore previous versions* of software. Thus, Sommerville [Som18, p. 865] defines a Version Control System (VCS) as an automated support of the version control process, which typically provides *centralized* (e.g., SVN[13]) or *distributed code repositories* (e.g., Mercurial[14], Git[15]). This way, developers can track single code differences (i.e., a set of added and deleted lines of code), which we refer to as *code hunks* (cf. [CS25, p. 24]).

A VCS handles changes to a code base by summarizing them as a *commit*, which creates a new version of source code and which is typically mapped with a *commit message*, a textual description by the developer summarizing the change [CS25, p. 36pp]. Commit messages can also link to identifiers from issue tracking systems, for instance, to denote that a certain commit fixed a previously reported software defect [SZZ05].

Moreover, it provides the ability to manage multiple code versions in parallel so that developers can work independently of each other. For this purpose, it applies a *branching mechanism*, which copies a certain version into a separate development branch without interfering with the original line of development (e.g., main branch) [CS25, p. 63pp]. Branches are typically labeled after a certain purpose (e.g., fix issue #42). After serving the purpose of the branch (e.g., fixing an issue), the changes are *merged*, for instance, back to the main branch. In case of conflicts (e.g., simultaneously changed source files), the VCS provides *conflict management* to resolve them. This mechanism enhances collaboration among multiple developers.

Finally, software developers use VCS to integrate continuous integration and deployment processes, such as automated building and testing of committed changes and deployment into production systems [Som18, p. 825pp].

3.4.2. Software Repository Mining

Term and Importance VCSs, together with the trend of *open-source development* and *advances in data mining*, support the research field of *Mining Software Repositories (MSR)* [KCM07, Vid22]. MSR denotes *the analysis of software repositories for various software engineering tasks to unveil characteristics and relationships of the software development process as well as to provide predictors for future software development* [KCM07, Vid22, Sch24]. One indicator of this trend is the increasing *International Conference of Mining Software Repositories (MSR)*[16], the major venue of MSR research present since 2004.

[13] https://subversion.apache.org/ last accessed: 2025/04/30
[14] https://www.mercurial-scm.org/ last accessed: 2025/04/23
[15] https://git-scm.com/ last accessed: 2025/04/30
[16] https://www.msrconf.org/ last accessed: 2025/05/02

Process and Analyzed Topics In general, MSR is not limited to VCS as a data source but also integrates data from issue trackers, archived e-mail communications, and Q&A webpages. [KCM07, Vid22]. In detail, Vidoni [Vid22] found that based on 146 MSR studies from 2005 to 2020, a majority of them analyze VCS (i.e., 67% based on GitHub and 8% on SourceForge) followed by multiple sources (e.g., VCS with Q&A pages) with 10%, and 7.5% on issue trackers (i.e., BugZilla). She complained that only 63% of the studies discuss their repository selection and data extraction processes, and even worse, 69% of the studies do not discuss data biases. Moreover, ethical and legal concerns arise in MSR research when creating and analyzing data from VCS systems [GK22].

Schwartz [Sch24, p. 28pp] derived a list of *18 different software engineering research topics* for which researcher apply MSR. Particularly, he discussed the topics of writing and refactoring code, code comprehension, code smell and quality analysis, general development aspects for entire projects, and human and team dynamics within software projects. Moreover, he identified six cross-cutting topics, one of them being the importance of machine learning techniques.

MSR research is also supported by tools [CSS13], with two prominently known techniques, named *BOA* [DNRN13][17], providing a domain-specific language to analyze snapshots of repository data, and *PyDriller* [SAB18][18], a Python library to interact programmatically with source code repositories.

3.4.3. API Evolution

API evolution refers to the changes of an API and, thus, in the respective library [BR22]. Since APIs are a part of a software product, these evolve similarly to ordinary software products [LGS21]. Having an intended change by API developers, [BR22] defines the following steps typically conducted when API changes:

1. Denote the *API deprecation*, meaning to mark API elements that should be no longer used, most presumed since they will be removed in subsequent API versions.
2. Create or update the *API documentation*, marking the changes as well as the deprecation.
3. *Informing the client developer* of the change and the deprecation of API
4. Client developers react to change, typically by *API migration* steps on the client side.

API Deprecation When API developers deprecate an API element, they flag this element to communicate to client developers that they shall not use it anymore. Such a flag can be the `@Deprecated`-annotation in Java[19]. The intention can be that this API element will be removed in subsequent versions or its maintenance will be ceased. Not properly deprecating APIs can cause *breaking changes*, for instance, directly removing an API element, which instantly causes errors in the client code [BVXH20].

Research supports API developers by providing guidelines and processes for proper API deprecation handling [EZG14, SAvDB18, MBS+19]. Moreover, Brito et al. [BHVR16] analyzed positive factors correlated with better deprecation messages by API developers,

[17] https://boa.cs.iastate.edu/ last accessed on 2025/05/02
[18] https://pydriller.readthedocs.io/ last accessed on 2025/05/02
[19] cf. https://docs.oracle.com/en/java/javase/22/core/how-deprecate-apis.html last accessed on 2025/02/14

for instance, small project size, large community, and more activities in the project. Kao et al. [KCJ22] introduced a technique to assess the potential impact of breaking changes, for instance, assessing the number of affected client applications. This impact data can help API developers to take special care for large breaking changes. Avoiding breaking changes, such as incompatibilities of the client code to a certain API version, can be achieved with principles like *backward compatibility*, which is successfully applied in the Android framework [HLW+18].

Create or Update API Documentation After updating the client code, API developers have to take care that the documentation evolves similarly so that client developers can review the changes between versions. For this purpose, Moreno et al. [MBP+17] suggested a technique to automatically generate *release notes*, which summarize the essential changes made between the two versions. Similarly, Dagenais and Robillard [DR14] presented a technique to identify the most relevant code changes, which an API developer can include in their documentation.

Informing the Client Developers A typical technique to inform client developers on deprecated APIs or breaking changes is the version number, namely, by applying so-called *semantic versioning* [RvV17]. However, since versioning is a manual process, API developers do not use the versioning consistently [RvV17]. For this purpose, automated techniques to detect possibly non-documented breaking changes in the client code were suggested [ZW16, VKC21].

API Migration Having a deprecated API, client developers should change the code so that no deprecated API elements are used, thus avoiding future breaking changes. However, research observed that still many deprecated APIs are used [SRB19, WLLC20]. Thus, different automated support for *API migration* was suggested. Three typically observed migrations are:

a) migration of client code from one API version to another (typically a subsequent) one;

b) migration of client code from one programming language to another;

c) migration of client code from one library, and thus API, to another one.

Much research was conducted in supporting the migration of code among different API versions (i.e., a)). A certain insight is that many migration issues could be handled by refactoring tools [DJ05, XS06]. Other techniques record and replay previous migrations [HD05], suggest API replacements or migration mappings [RBK+13, HCP+21], mine migration rules [SJM08, NNW+10, WGAK10, MWZM12, LS18, XDM19, LSC22, WY22, ZDAT22, RML+23], or directly update the source code [HTL+21a, HTL+21b]. Early techniques to migrate client code among programming languages (i.e., b)) used API mappings [ZTX+10], while more modern machine learning-based techniques leverage an embedding to relate API elements [NNPN17, CXLX21b, ZWX+23]. For cross-library migration (i.e., c)) similar techniques were used for finding API mappings [TFB13, DND+25]. There exist other specialized migration techniques, such as migrating tests of graphical user interfaces [QZW19] or techniques fixing deprecated client software during runtime [DPZ+22].

Moreover, Arvedahl [Arv18] described a technique applied in the industry that avoids the usage of deprecated API elements by client developers.

3.5. Frequent Pattern Mining

A number of static API misuse detection techniques build on the data mining technique of *Frequent Pattern Mining (FPM)*. Particularly, these techniques conduct FPM on API usages (e.g., code examples represented as token streams or ASTs) to obtain patterns as specifications of the API usage. This technique is also known as *(API) specification mining*. In this section, we dive into the state of the art of FPM first, to understand the terminology used in FPM and advances made so far, and second, to assess the FPM techniques applied for API specification mining. This section is mainly based on the textbook on Frequent Pattern Mining edited by Aggarwal and Han [AH14] and particularly, the chapters on the introduction to FPM [Agg14] (i.e., Chapter 1), FPM algorithms [ABH14] (i.e., Chapter 2), pattern-growth techniques [HP14] (i.e., Chapter 3) interesting patterns [VT14] (i.e., Chapter 5), sequential pattern mining [SWH14] (i.e., Chapter 11), and graph pattern mining [CYH14] (i.e., Chapter 13).

3.5.1. Problem Statement of Frequent Pattern Mining

FPM targets the inference of *interesting* relationships in the form of patterns between *items* in a given *database*. This rather generic description can be further specified by zooming into the emphasized characteristics of FPM.

Interesting Patterns First, FPM focuses on *interesting* patterns. Interestingness is a highly subjective term according to the use case at hand for which FPM is conducted. For instance, for the well-known example of market basket analysis (i.e., determining groceries frequently bought together) [Agg14, VL16a], the interestingness can be measured by the frequency (i.e., how often certain groups of groceries are bought together) as well as by other metrics, such as which groups of groceries obtain the highest profit. This frequency may be interesting for owners of grocery stores to optimize their supply and, thus, their profit. According to the topic of API specification mining, interesting patterns represent correct API usages that accurately discriminate correct from false applications of an API [WN05, AX09]. Even though many FPM techniques measure interestingness via the frequency of items within a given dataset, frequency yields not always the aspired patterns as seen in generic pattern mining use cases [VT14] as well as in API specification mining [LW09, LL15, ANN+19a]. We discuss different interestingness measurements in Section 3.5.3.

Items Second, FPM considers different abstractions of *items* for which patterns should be inferred. These items, as well as their patterns, represent different kinds of features, such as groceries bought in a store [Agg14] or API method calls in the source code [RBK+13]. But items and patterns are also represented in different kinds of data structures, such as itemsets, sequences, or graphs [RBK+13, SWH14, CYH14]. Different data structures are applied to entangle certain mandatory or aspired relationships into the mining process [Agg14]. For instance, source code has a strict structure defined by the grammar of the respective programming language, which can be represented by an AST or a CFG. This structure enforces the inference of syntactically correct patterns (i.e., sub-ASTs or sub-CFGs). This benefit, however, is at the expense of transforming the items into the respective data structure as well as processing the mining with more sophisticated algorithms.

Database Finally, FPM is conducted on a *database* that contains so-called transactions of items [Agg14]. Depending on the applied data structure, this database consists of differently structured transactions (e.g., a set of itemsets, sequences, or graphs). The database has two important properties. First, *if a pattern is not present in a database, standard FPM techniques cannot infer it* [Agg15][20]. Second, if the *interestingness measurement depends on some characteristics of the database, such as its size [VT14], the inference of patterns is influenced by the database.* This influence means that having a meaningful database from which to mine patterns is crucial obtaining interesting patterns.

We summarize the problem statement of FPM in the following formal description: The goal of FPM is to extract the pattern set P from a database D consisting of transactions $T_1, \cdots, T_n$ such that for an interestingness measurement function w.r.t. D denoted f_D and a user-defined minimal interestingness threshold F_{min} it holds $\forall p \in P : f_D(p) \geq F_{min}$[21].

3.5.2. Mining Algorithms

A simple instantiation of the problem described above is the frequent itemset mining, whose description we base on Aggarwal et al. [ABH14]. Here, the database transactions represent single sets of items, and the interestingness measurement (i.e., function f_D) is based on frequency, namely, the support of a pattern p. The support denotes the proportion of transactions $T_1, \cdots, T_n$ in the database D that contains the pattern p. Thus:

$$support_D(p) = \frac{|\{T_i | p \in T_i \wedge T_i \in D\}|}{|D|}$$

The output of the mining algorithm is the set of all $p_j \in P$, which support achieves a minimal threshold, namely,

$$support_D(p_j) \geq min_{support}.$$

We refer to $min_{support}$ as the *minimal relative support* while $|D| \cdot min_{support} = min_{support_{abs}}$ denotes the *minimal absolute support* (i.e., the minimal absolute number of transactions in D containing pattern p).

Naive Algorithm A naive algorithm is to enumerate all possible candidate patterns (i.e., all possible non-empty sub-itemsets among the transactions in D) and check whether each candidate satisfies the minimal support condition. This algorithm, however, is not feasible since, for the itemset mining, there exist at least $2^m - 1$ possible candidates [VT14, VL16b][22], where m denotes the number of distinct items in the database D. Then, for each candidate at most $|D| - min_{support_{abs}} + 1$ entries need to be checked (i.e., to denote whether a candidate is infrequent). Having a large number of transactions (i.e., $|D|$) together with a typically low $min_{support}$ and/or a large number of distinct items (i.e., m) hinders efficient pattern inference.

[20] assuming patterns are not altered afterward independently of the database

[21] This definition is similar to the formalization of theory mining [MT97]. However, in theory mining, the condition denoting interestingness is generalized to be a boolean formula. In our definition, we assume that interestingness is an ordinal value, which is sufficient for our use case.

[22] power set of all item sets minus the empty item set

Apriori Algorithm A breakthrough was achieved by Agrawal and Srikant [AS94] as well as Mannila, Toivonen, and Verkamo [MTV94], who, independently of each other, developed the *Apriori algorithm* to efficiently mine association rules. Association rules are based on a database of itemsets D and have the form $A \rightarrow B$ with $A \cap B = \emptyset$ where A and B denote sub-itemsets from D. A sub-problem of mining association rules is the above-stated itemset mining problem. After computing the support of each itemset (e.g., A and B), one can easily obtain the association rules by computing the confidence of a rule $A \rightarrow B$ denoted as the ratio of $support_D(A \cup B)$ and $support_D(A)$ and selecting those rules exceeding a minimum confidence. For support computation, they effectively pruned the candidates' space by leveraging the anti-monotonic [Leu09] or *Apriori property* of the *support* function, namely, if pattern p is a subpattern of p' (i.e., in case of itemsets $p \subseteq p'$) and $support_D(p) < min_{support}$ then $support_D(p') < min_{support}$ holds. The Apriori algorithm enumerates the candidates beginning with itemsets of size $k = 1$ (i.e., single items) and selects those that satisfy the minimal support condition. These candidates form the set of *frequent* 1-patterns, while the others are removed for further analysis since their extensions (i.e., supersets) would not produce any frequent patterns. To produce the next $k + 1$ candidates, we join all pairs of frequent k-patterns (i.e., in itemset mining using the union operation). This procedure is iteratively repeated until no further frequent patterns are found [ABH14].

Closed and Maximal Patterns Even though efficient, this standard algorithm has some drawbacks [ABH14]. First, often the *same patterns* are generated and tested. Second, many patterns represent subparts of even larger patterns and thus tend to be redundant. Third, for each support condition check, one needs to process almost the complete database. These drawbacks impact the practical applicability since the mining produces a too-large set of possible uninteresting redundant patterns, as well as frequent re-processing of the database, increasing the mining time.

To reduce the number of redundant patterns, the notion of *closed* [PBTL99a] and *maximal* [Bay98] patterns was introduced. Closed patterns denote those patterns for which no frequent superpattern exists having the same support. For itemset mining, a closed pattern is denoted as a pattern $p \in P$ (note that all $p \in P$ satisfy $support_D(p) \geq min_{support}$) is closed if $\nexists p' \in P : p \subset p' \wedge support_D(p) = support_D(p')$. Instead, maximal patterns denote closed patterns for which all superpatterns are not frequent at all. Formally, this means $p \in P$ is maximal if $\nexists p' \in P : p \subset p'$. This way, mining only closed or maximal patterns reduces the number of results significantly since they encapsulate many or all of their subpatterns [ABH14]. Moreover, several algorithms leverage these properties by ignoring non-closed and non-maximal patterns in the mining process (e.g., Close [PBTL99b], CLOSET [PHM00], MaxMiner [Bay98]). Details on such algorithms are described by Aggarwal et al. [ABH14]. Note that while for closed patterns, we can derive the support values for each subpattern, maximal patterns lose this information [ABH14].

Further Optimizations There exist several optimizations of the Apriori algorithm mainly to scale and speed up the mining process, for instance, by simplifying the transactions in each iteration step (e.g., AprioriTiD [AS94]) or by reusing the counting work for the support (e.g., with direct hashing and pruning, DHP-algorithm [PCY95]). Another group of efficient mining algorithms is formed by *pattern growth algorithms* [HP14]. These algorithms get rid of the tedious candidate generation by applying sophisticated data structures to track

and reuse the counting work and infer patterns recursively by building database projections of the original database (e.g., frequent pattern growth or FP-growth algorithm using FP-trees [HPY00]).

Sequence and Graph Mining Finally, significant effort was put into the development of algorithms handling more complex data structures than itemsets. We consider algorithms for sequence [SWH14] and graph mining [CYH14] since these were mainly used in the domain of API specification mining [RBK+13].

According to Shen et al. [SWH14], Frequent Sequence Mining (FSM) is defined as follows: A sequence denotes an ordered list of itemsets s_j, namely, $\langle s_1, \cdots, s_l \rangle$. A sequence $\alpha = \langle a_1, \cdots, a_n \rangle$ is denoted a subsequence of $\beta = \langle b_1, \cdots, b_m \rangle$ if

$$\exists j_1, \cdots, j_n : 1 \leq j_1 \leq \cdots \leq j_n \leq m, \text{ such that } \quad a_1 \subseteq b_{j_1}, \cdots, a_n \subseteq b_{j_n} \\ \text{(i.e., denoted as } \alpha \sqsubseteq \beta).$$

This definition allows some intermediate itemsets in a sequence to be skipped in subsequences (e.g., removing noise in long sequences). The transactions in the database of FSM represents sequences together with their identifier. The absolute support of a sequence α denotes the number of sequences s in D that satisfy $\alpha \sqsubseteq s$. The relative support (i.e., $support_D(\alpha)$) is defined similarly as stated above by dividing the absolute support by the number of all entries in D. The goal of FSM is to find all pattern sequences α that satisfy

$$support_D(\alpha) \geq min_{support}.$$

Previous work performed *FSM* by applying the Apriori-like algorithms on sequences (e.g., AprioriAll [AS95], GSP [SA96], PSP [MCP98]). These algorithms create new candidate sequences by joining frequent sequences overlapping in all but the first and last element. Moreover, there exist also pattern-growth-based algorithms (e.g., FreeSpan [HPM+00], PrefixSpan [PHM+01]) with some extensions to mine closed sequence patterns (e.g., CloSpan [YHA03], BIDE [WH04]).

According to Cheng et al. [CYH14], *Frequent Graph Mining (FGM)* aims to find frequent subgraphs in a set of graphs[23]. In detail, assume a graph g, its vertex set $V(g)$, and its edge set $E(g)$, as well as its label function l. Then, g is a subgraph of g' (i.e., $g \subseteq g'$) with g' having the label function l' if there exists a subgraph isomorphism. A subgraph isomorphism denotes a function

$$f : V(g) \to V(g'), \text{ such that } \quad \forall v \in V(g) : \quad l(v) = l'(f(v)) \\ \text{and} \\ \forall (u,v) \in E(g) : \quad (f(u), f(v)) \in E(g') \wedge \\ l(u,v) = l'(f(u), f(v)).$$

FGM is based on a database D of graphs $g_1, \cdots, g_n$ to retrieve graph patterns g_p, which form a supporting set $D_{g_p} = \{g_i | g_p \subseteq g_i, g_i \in D\}$ and which satisfy

$$\frac{|D_{g_p}|}{|D|} = support_D(g_p) \geq min_{support}.$$

[23] Graph mining can also be considered as finding frequent subgraphs in a single, large graph (cf. [CYH14]). Since this problem definition is usually not used in API specification mining, we do not detail it in this work.

This definition means that finding frequent graphs requires frequently solving the general subgraph isomorphism problem, which is known to be NP-complete [Epp99]. Moreover, when performing Apriori-like algorithms, we need to generate candidate graphs by joining frequent subgraphs by extending them via nodes or edges. This generation, however, denotes an exponential increase in the candidate space and, thus, the mining effort [CYH14]. Known algorithms to solve the FGM problem are Apriori-based (e.g., AGM [IWM00] and FSG [KK01]), pattern-growth-based (e.g., gSpan [YH02]), as well as different specializations to mine closed graph pattern (e.g., CloseGraph [YH03]).

3.5.3. Interesting Patterns

We discussed that one main goal of FPM is to infer a set of interesting patterns, namely, patterns suitable for a certain use case. As discussed in the previous section, many mining techniques apply the support (i.e., frequency) measurement for this purpose. This approach is reasonable since it exhibits the anti-monotonic or Apriori property, which allows effective pruning of the candidate search space. However, support does not always yield interesting patterns or produces many similar and, hence, redundant patterns.

This behavior has also been demonstrated in the domain of API specification mining and API misuse detection. Le and Lo [LL15] discussed in rule-based API specification mining (i.e., finding association rules for API elements) that support and confidence do not represent optimal measurements when comparing 38 different measurements. Amann et al. [Ama18, ANN+19a] observed problems when ranking violations of patterns (i.e., API misuses) based on the respective pattern's support, namely, these violations are not necessarily correct.

To target this issue in FPM, many different interestingness measurements have been suggested [VT14]. However, most of these do not exhibit the Apriori property. Thus, it is common practice, first, to mine frequent patterns using support and then to rank and filter those patterns using one or multiple interestingness measurements. Vreeken and Tatti [VT14] distinguish interestingness between *absolute* and *advanced* measurements.

Absolute measurements use mainly the pattern itself to assess interestingness without a broader context. These measurements target the previously introduced *closed* and *maximal* patterns but also pattern types such as *free* (i.e., no frequent sub-patterns with equal support) [BBR03] or *robust* patterns (i.e., patterns inferred from multiple, random subsets of the database) [TMC14]. More complex techniques apply Markov-Chain Monto-Carlo sampling [BMG12] as well as tile mining (i.e., in itemset mining tiles represent frequent areas among the rows of transactions and columns of items) [GGM04].

Due to their limited context knowledge about other patterns, these algorithms do not necessarily find statistically different patterns (i.e., they do not exhibit a notion of surprise). This issue is targeted by advanced measurements. Particularly, these measurements represent to which degree a pattern deviates from the expected statistical model. A simple function is the *lift*, which is denoted according to an association rule $A \rightarrow B$ as:

$$lift(p) = \frac{support_D(A \cup B)}{support_D(A) \times support_D(B)}$$

The $lift$ compares the frequency of a rule to the frequency suggested by the independence model and thus describes how surprising the rule is according to a uniform distribution. This same notion has been further extended by starting with a simple statistical model and refining it with each newly inferred pattern. This way, the statistical model learns to

assess whether new patterns are significantly different from previously inferred ones. Such models use a Bayesian model [JS04] or the principle of maximum entropy [TM10].

3.5.4. Databases for Mining

General As stated before, a database is a crucial element to successfully infer meaningful patterns. Thus, the creation of such a database requires special care. Aggarwal [Agg15] discusses three essential data preparation steps for data mining:

1. *extracting features as applicable data structures*, namely, extraction from raw data and its abstraction as known data structures (e.g., sequences or graphs)
2. *cleaning the data*, namely, filtering erroneous data or estimating missing data
3. *reducing* data size, *selecting* data features, and/or *transforming* data, for instance, data reduction by sampling, feature selection by summarizing correlated data via redefining features (e.g., using principle component analysis - PCA), and transformation by changing data structures (e.g., transforming complex data into less complex, multidimensional data)

Especially the third step strives for the two goals of (1) a smaller dataset that is more likely efficiently processed by more precise mining algorithms and (2) a purpose-related data selection and transformation, which can increase the quality of data mining results.

More generally, the whole data mining process, including the preparation of a meaningful database, has been summarized in the industrial process *CRISP-DM (i.e., **CR**oss-**I**ndustry **S**tandard **P**rocess for **D**ata **M**ining)* [CJCaK+00, MPCOF+21].

This process consists of several phases, as depicted in Figure 3.2. We focus on the following four specific components of that process (emphasized in gray) that we find important for creating a meaningful database based on the discussions by Chapman et al., Schröder et al. and Martínez-Plumed et al. [CJCaK+00, SKG21, MPCOF+21]:

- (A) *business understanding*
- (B) *data understanding*
- (C) *data preparation*
- (D) *data*

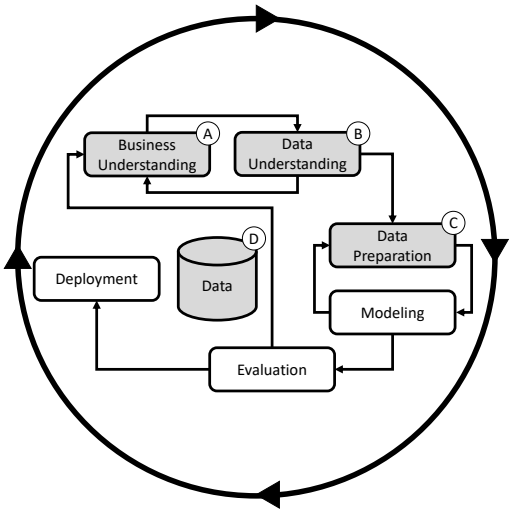


Figure 3.2.: CRISP-DM according to Martínez-Plumed et al. [MPCOF+21] with focus on the steps for producing the database (i.e., in gray)

Business understanding (cf. (A)) considers the specific domain in which a data mining project is planned and thus defines its goals and provides justification for certain data mining processes and techniques targeting this domain [CJCaK+00, SKG21].

Within the *data understanding* (cf. (B)) step, a first data collection is conducted, which is accompanied by a discussion of potential data sources as well as the provision of descriptive

information on the data, for instance, to detect data issues (e.g., data biases) [CJCaK+00, SKG21]. According to CRISP-DM, business and data understanding are conducted in an iterative manner.

The *data preparation* (cf. Ⓒ) step denotes processes and techniques to obtain the final dataset. In detail, it encompasses the cleaning of data as well as several mechanisms for selecting and transforming the data. It is usually processed in an iterative manner together with the modeling step (e.g., including training a machine learning model or conducting FPM) [CJCaK+00, SKG21].

Martínez-Plumed et al. [MPCOF+21] also discussed that modern applied data science can differ from these strict steps denoted by CRISP-DM towards more loosely combined components. In detail, they suggest the inference of several different data science trajectories consisting of these steps together with further activities depending on the specific data science task. Nevertheless, business and data understanding, as well as data preparation, are still important steps in their view.

Moreover, they detailed the *Data* block (cf. Ⓓ) with four important activities:

- *acquisition*, namely, collecting or producing relevant data
- *simulation*, namely, creating data via simulation, particularly for complex systems
- *architecting*, namely, exploring different layouts of the data (i.e., logically and physically) and using different data sources
- *release*, namely, publishing and making the data accessible

API-Specific Implications As a consequence, it is important for a successful generation of datasets representing API usages for API specification mining to adhere to these previously discussed steps. In detail, we require a business understanding, namely the domain of APIs and libraries, as discussed in Sections 3.1.1 and 3.1.3.

Moreover, a data understanding is required, particularly with specific data structures representing API usage, as discussed in Section 3.2.2.

Regarding the data preparation, we discussed potential data sources with code repositories as well as techniques such as code search and repository mining (cf. Sections 3.3 and 3.4.2). This way, we have to critically review existing data sources of API usages and misuses, together with their preparation steps, and discuss potential implications on subsequent applications (e.g., API misuse detectors).

Regarding the dataset itself, we focus on publically available datasets and provide our datasets of API usage in respective replication packages.

3.5.5. API-Specific Usage Pattern Mining

Amann et al. [Ama18, ANN+19b] presented a mining algorithm to infer frequent AUGs (cf. Section 3.2.2) as API usage patterns.

Mining Algorithm In essence, this algorithm applies the *Apriori algorithm with closed graph mining* [Ama18, ANN+19b]. The miner iteratively grows patterns from single AUG nodes to frequent subgraphs using the neighbor edges from single transaction AUGs from a database (i.e., AUGs of API usage examples). The extension of pattern candidates is obtained by clustered isomorphic subgraphs, which makes closed pattern mining much

more efficient. Since the general problem of subgraph isomorphism is known to be NP-complete [Epp99], they apply a graph vectorization named *Exas vectors* [NNP+09a]. The assumption is that if the hash values of two graph vectorizations are equal, both graphs are considered isomorph. While this heuristic over-approximates the number of isomorphic extension graphs, they found promising results for their miner, particularly for API misuse detection [ANN+19b]. They determine the support of each pattern candidate and report those patterns exceeding the minimal support value (i.e., $min_{support_{abs}}$).

Configurations Amann et al. [Ama18, ANN+19b] applied different support definitions[24], which can be configured, namely

- *within-method support* - counting all pattern occurrences
- *cross-method support* - counting all methods containing a certain pattern candidate
- *cross-project support* - counting all projects containing a certain pattern candidate

Moreover, one can configure the *minimal and maximal pattern size* (i.e., number of nodes) to further restrict the number of pattern candidates during mining. Finally, they provided and tested *various ranking mechanisms* for the reported patterns, going beyond pure support.

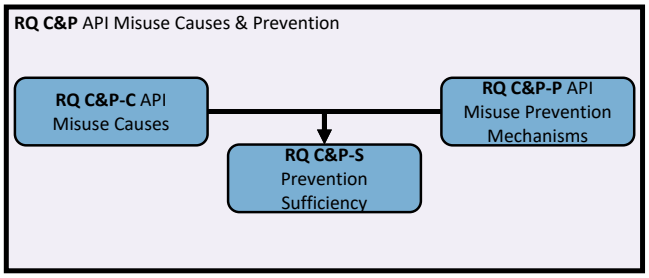
[24] based on the code base from MUDetect https://github.com/stg-tud/MUDetect last accessed: 2023/07/11

4

API Misuse Root Causes & Prevention

In this chapter, we analyze the root causes and prevention mechanisms of Application Programming Interface (API) misuses. Preventing known root causes of API misuses would avoid the detection and the repair, which we separately discuss in the chapters 5, 6, and 7.

4.1. Methodology and Structure

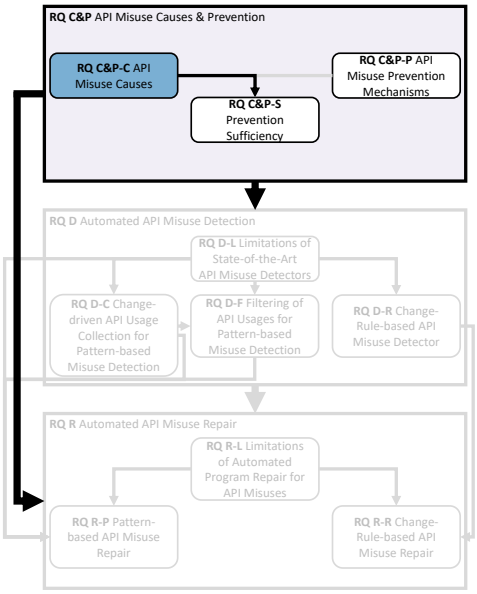


We tackle the challenge of missing an overview of root causes and their prevention mechanisms of API misuses, and thus, we target **RQ C&P**. Particularly, we determine the state-of-the-art on API misuse root causes (i.e., **RQ C&P-C**) and their prevention mechanisms (i.e., **RQ C&P-P**). Subsequently, we combine those results to assess whether state-of-the-art research on prevention sufficiently targets all found API misuse root causes (i.e., **RQ C&P-S**).

We answer these questions by applying Systematic Literature Review (SLR) and qualitative research methodologies from the field of software engineering research according to the ACM SIGSOFT Empirical Standards [RAB+20]. In particular, we use the methodologies from guidelines on SLR [KC07, Woh14] as well as on qualitative research [Fli14] when collecting and summarizing the state-of-the-art (i.e., for research questions **RQ C&P-C** and **RQ C&P-P**). Finally, we determine research opportunities by summarizing the results in a structured way.

Subsequently, we target **RQ C&P-C** in Section 4.2 by first stressing the limitations of related surveys (cf. Section 4.2.1), presenting our concrete methodology and our results on the overview of API misuse root causes (cf. Section 4.2.2), as well as their limitations in the threats to validity (cf. Section 4.2.3). Section 4.3 targets the research questions **RQ C&P-P** and **RQ C&P-S**. In a similar structure as in Section 4.2, we provide the overview of and the difference to previous surveys on prevention (cf. Section 4.3.1), describe our methodology and the results of the SLR (cf. Section 4.3.2), and thus the answer to **RQ C&P-P**. Based on this, we discuss limitations and further research abilities by combining the results on API misuse causes and their prevention (cf. Section 4.3.3). We discuss the limitations of these results in Section 4.3.4. We summarize the main results and answers to sub-research questions in Section 4.4.

4.2. API Misuse Causes



In this section, we discuss the question of the root causes of API misuse, namely **RQ C&P-C**. For that purpose, we conducted an SLR process to collect related research, applied open coding to analyze the causes of API misuses and mapped them to their applied scientific methodologies to assess their scientific evidence.

4.2.1. State-of-the-Art Meta-Analyses on API Misuses Causes

There exist many surveys (i.e., SLR and mapping studies) based on primary studies concerned with the root causes of API misuses. However, those SLRs and mapping studies lack a complete overview of the state of the research. Table 4.1 illustrates this lack by the summary of those literature analyses.

We found that those surveys typically concentrate on a single API misuse causes like:

- issues caused by API evolution, such as API deprecation [LGS21, BR22]
- API documentation [NAP18, CVG19]
- API usability [Zib08, BFHM12, RTP19, PDHR23]

Exceptions were works by Ochoa et al. [OHG+25] and McGregor [McG23], in which both considered multiple root causes.

Except for the literature review by Zibran [Zib08], all studies documented their process to retrieve related publications. We observed that our analysis (cf. gray row in Table 4.1) obtained comparable results with respect to the analyzed time frame and number of analyzed papers. An exception regarding the number of analyzed publications was the study by Lamothe et al. [LGS21], which broadly investigated the general topic of API evolution having softer rules for inclusion, for instance, no restriction on the type of study.

In more detail, next to the considered root causes, the studies differed in the methodology and, thus, their results. Lamothe et al. [LGS21] provided a quantitative overview of research topics related to API evolution (i.e., maintenance, usability, and others), contribution types of API evolution studies (i.e., mostly tools and techniques as well as empirical studies), and the applied evaluation methodologies and metrics. Similarly, Bonorden and Riebsch [BR22] analyzed the topic of API deprecation and provided a quantitative overview of profiteers of these studies (i.e., mostly human stakeholders), contribution types of the studies (i.e., mostly descriptive knowledge and technical solutions), research strategies (i.e., mostly data-driven), types of analyzed APIs, and analyzed aspects of deprecation (i.e., mostly client developer-side than API developer-side).

The root cause of API documentation was the focus of the literature review by Nybom et al. [NAP18]. They provided a quantitative overview of techniques and approaches to generate and support API documentation, as well as evaluation metrics and techniques used in the analyzed studies. Moreover, Cummaudo et al. [CVG19] inferred a taxonomy

Table 4.1.: Overview of related systematic literature analyses on API misuse causes compared to ours

Misuse Causes	Reference	#Publications	Time Frame
API evolution	[LGS21]	369	1994-2020
	[BR22]	36	2005-2021
API documentation	[NAP18]	36	2000-2016
	[CVG19]	21	NA-2019
API usability	[Zib08]	NA	NA
	[BFHM12]	28	2004-2011
	[RTP19]	47	1998-2018
	[PDHR23]	65	1974-2021
client developer, API usability, API documentation	[McG23, p. 13pp]	73	NA-2023
API installation, API code, API evolution, API documentation paired with licensing and domain-related issues	[OHG+25]	35	1996-2023
General	**Ours**	**65**	**2007-2023**

and provided a quantitative overview of research directions on API documentation support depending on research subjects and evaluation methodologies.

The aspect of API usability was investigated in the literature review by Zibran [Zib08], particularly factors influencing them, which were summarized in a qualitative way. However, their review has only limited information on the review process and their selected publications. This missing information hardens the traceability of their results. Burns et al. [BFHM12] also considered API usability with a focus on recommendations on design as well as the applied evaluation metrics and techniques of the analyzed studies. Their results were provided as a qualitative summary. Instead, Rauf et al. [RTP19] gave a quantitative view of API usability with a focus on applied methodologies for API usability analysis (i.e., more empirical than analytical methodologies), the kind of contribution of the studies (i.e., mostly new evaluation approach), and a qualitative overview of usability factors among the studies together with their evaluation metrics. Patnaik et al. [PDHR23] also provided a quantitative overview of recommendations implying API usability but with a focus on security APIs. Particularly, their results emphasized the differences between software engineering and security research regarding recommendations for API usability and the limited evaluation of security-related studies.

The literature review by McGregor (cf. [McG23, p. 13ff]) found misuse causes regarding the client developer (i.e., API and domain learning issues), the API usability, and the API documentation. In detail, he also focused on security APIs and, particularly, mapped the root causes onto potential security threats. A detailed enumeration of single sub-causes for usability and documentation related to the surveyed publications was given in Table 2.2 (cf. [McG23, p. 16]) and Table 2.3 (cf. [McG23, p. 22]) of his dissertation.

In their recent survey, Ochoa et al. [OHG+25] analyzed primary studies on API usages, misuses, and repair techniques. Within their survey, they also considered root causes, however, they focused on technical issues, particularly on the installation (e.g., configuration), the code (e.g., software defects in libraries), the API documentation, the API evolution

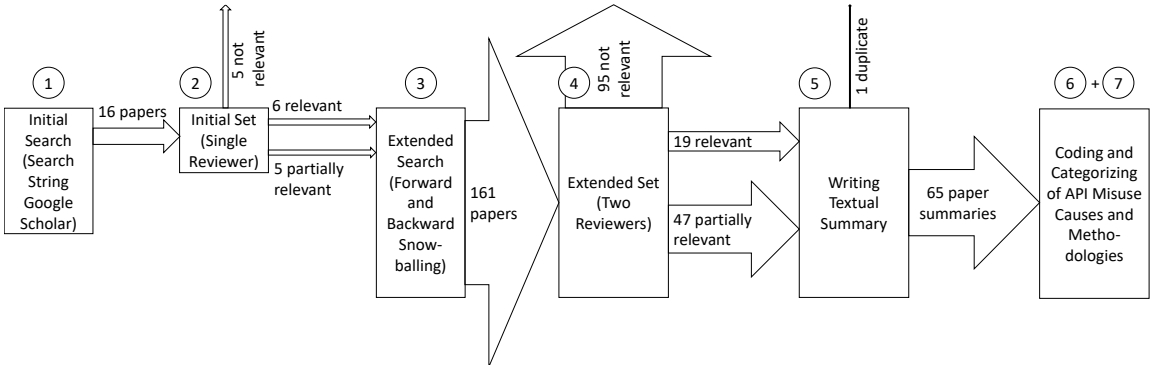


Figure 4.1.: Overview of SLR process for meta analysis of API misuse causes and the research methodologies

(e.g., due to changing requirements in client code) as well as related issues due to the license and the specific domain. However, their discussion does not encompass human factors, such as our survey.

In general, these studies give overviews of certain aspects of API misuse causes, critically review the applied methodologies, and pave the way for further research directions. While our conducted analysis has a more general view on the topic of API misuse causes, the present results are valuable and will be included in our final discussion of root causes. Moreover, the analyses served as the basis for the prevention mechanisms literature review in Section 4.3, except for the work by Ochoa et al. [OHG+25], which was retrieved after the analysis.

4.2.2. Meta-Analysis of API Misuse Causes

Methodology By the meta-analysis, we determined the current state of research on the root causes of API misuses. For this purpose, we analyzed current publications on this topic by applying an SLR methodology [KC07]. Note that our goal *was not* to retrieve an overview of *all* existing literature regarding API misuse causes but to summarize the general body of knowledge on API misuse causes. This way, we did not conduct an exhaustive but a sufficient search identifying the most relevant research on root causes of API misuses. Moreover, we also incorporated an analysis of the empirical methodologies applied to find the root causes. This analysis allowed a better assessment of whether certain root causes were found by different methods, which strengthened their validity. This way, we also identified potential research directions, for instance, root causes, which could be analyzed using a different methodology. We did this by applying qualitative analysis methods, namely, open coding (cf. [Fli14] p. 373ff) and situational analysis (cf. [CFW24] p. 369ff).

An SLR is characterized by a systematic process defining the search, the selection criteria, the information extraction, and the analysis process [KC07, Woh14]. Additionally, Krüger et al. [KLvN+20] stressed that many SLR studies suffered from a lack of information for replication, for instance, the applied guidelines and deviations from these guidelines. Thus, it is mandatory to precisely describe the SLR process, which we orientate towards the guideline by Kitchenham and Charters [KC07]. An overview of our process, together with a number of identified relevant papers, is depicted in Figure 4.1.

In essence, Kitchenham and Charters [KC07] suggest three main steps to conduct an SLR

1. Planning
2. Conducting
3. Reporting

The first ***Planning*** step targets whether an SLR is necessary, and to define the research question. We already discussed the necessity in Section 4.2.1 and clarified our research question (i.e., **RQ C&P-C**). Further sub-steps target the conductor of the SLR, which is the author of this thesis (i.e., subsequently referred to as the *main author*), together with another software engineering researcher, as well as determining and validating the review protocol. We define the process in the subsequent paragraphs. However, we did not conduct an intensive evaluation of this process since we aligned our steps to the aforementioned guideline. Moreover, we planned different steps to find relevant literature to target internal validity threats and discussed the results with software engineering experts from science and industry to target external validity threats. We detail this in Section 4.2.3.

In the ***Conducting*** step, according to Kitchenham and Charters [KC07], an SLR requires the definition of a search strategy, which we present in steps ① and ③ of our SLR process. Then, we define the selection criteria in steps ② and ④, which also encompass a required quality check of the studies. Step ⑤ describes how information is extracted, and steps ⑥ and ⑦ denote the analysis and, thus, the synthesis from the relevant literature.

Finally, the ***Reporting*** step denotes the format of presenting the results as well as its dissemination. We present the results of our SLR analysis in the *Result* paragraphs.

① *Initial Search.* We applied the scholar search engine Google Scholar[1] to conduct the initial search. We chose Google Scholar since it was assumed to be less dependent on publisher bias [Woh14]. However, search results may differ in the number of search results depending on the search day, as has been observed with other search engines [KLvN+20]. After a process of trial and error to find relevant publications, we derived the following final search string:

```
"cause" OR "reason" AND "API Misuse" AND "Empirical Study"
```

In the initial search, we filtered the results in the time frame from 2000 up to 2020 and conducted our search on September 20th, 2020. We obtained a set of 176 publications, from which the main author selected 16 potentially relevant papers based on their title.

② *Initial Set.* Based on these 16 publications, we applied the following additional selection criteria, by which we also ensure the quality of the publications:

1. The publication has gone through an elaborated review process (i.e., full paper[2] from a conference or a journal or a reviewed dissertation thesis);
2. The publication is written in English;
3. It is published between 2000 and 2020 (in a later search, namely step ④, up to 2023);
4. In the publication, the authors analyzed API misuses and their potential root causes;

[1] https://scholar.google.de/ last accessed: 2024/10/08
[2] note that this excludes short papers and idea papers

5. In the publication, the authors analyzed real-world API misuses;
6. In the publication, the authors used a valid empirical study methodology (i.e., qualitative or quantitative method).

We checked these criteria in a manual process by reading the potentially relevant papers:

- If the reviewer ascertained that the publication satisfies all criteria, it is denoted as *relevant*.
- In case one of the criteria could not be clearly answered with a 'yes' by the reviewer, it is denoted as *partially relevant*. This situation mostly denoted whether the issues discussed in the paper represent API misuses or the study considers only a very restricted aspect of API misuses.
- Otherwise, it is denoted as *not relevant*.

For this initial set, the main author operated as a single reviewer and assessed the publications as follows:

- 6 as *relevant*
- 5 as *partially relevant*
- 5 as *not relevant* from which one is a related literature review

③ *Extended Search.* We conducted an extended search, based on the initial set of relevant papers determined in the previous step, to cope with inconsistent search results of search engines (i.e., different results depending on the time of search). In detail, we applied an iterative approach named *snowballing* [Woh14]. Particularly, we used *backward snowballing* to find publications that were cited by papers denoted as *relevant* or *partially relevant* in the previous step. Backward snowballing was done by consulting the reference list of the respective publication. In contrast, we also applied *forward snowballing*, which aimed to determine papers that cited those publications, found to be *relevant* in the previous step. We achieved this by applying the 'cited by' functionality of Google Scholar[3]. We restricted forward snowballing to *relevant* papers to keep the number of search results manageable.

For all retrieved papers, we selected a set of potentially relevant papers based on their title to reduce initial manual effort. Then, we applied the same criteria as discussed in step ② with the difference of collecting publications up to publication year 2023 (i.e., criteria 3.). This collection was done by the main author in an iterative manner, meaning applying the above-mentioned snowballing procedure for all newly obtained *partially relevant* (i.e., only backward) and *relevant* (i.e., both forward and backward) papers until no further *partially relevant* or *relevant* papers were found. This way, we found 157 publications.

④ *Extended Set.* To avoid reviewer bias, a second software engineering PhD student, independently (i.e., without knowing the assessment of the first reviewer but with the same criteria), read and assessed those 157 publications. Since one paper was not accessible during the first review but in the second review and was found to be *relevant*, an additional snowballing iteration was conducted. This iteration found four additional papers, summing

[3] cf. https://scholar.google.com/intl/de/scholar/citations.html#citations last accessed: 2024/10/08

Table 4.2.: Result of the independent assessment of reviewers on publications before in-person discussion. The cells highlighted represent the publications which were discussed in-person.

reviewer 1 \ reviewer 2	*relevant*	*partially relevant*	*not relevant*
relevant	11	15	13
partially relevant	3	10	14
not relevant	6	14	75

up to 161 publications. In Table 4.2, we depict the number of decisions in comparison to the decision of the respective other reviewer. For instance, reviewer 1 found in sum 39 *relevant* papers, of which 15 were assessed as *partially relevant* by reviewer 2. We observed that in 96 cases, both reviewers had a perfect match. In 18 cases, reviewers did not agree on whether the papers were *relevant* or *partially relevant*, which was not critical since both kinds were considered in the final qualitative analysis. In the final decision, we assessed them as *partially relevant*. However, in 47 cases (i.e., highlighted in Table 4.2), one reviewer decided that the paper was *not relevant* while the other reviewer did so. Thus, we discussed and resolved these conflicts in a single in-person meeting and agreed on the following assessment:

- 19 *relevant* papers
- 47 *partially relevant* papers
- 95 *not relevant* papers

⑤ *Writing Textual Summary.* The main author read all 66 papers (i.e., 19 *relevant* and 47 *partially relevant*). While reading, we noticed a single publication, which was a journal extension of a conference paper already included in the extended set. Since it did not provide any additional information on the root causes of API misuses, we excluded it from further analysis (i.e., one *partially relevant* paper), leaving *19 relevant and 46 partially relevant papers*. Then, we summarized the papers by having a special focus on the methodology as well as on the root causes identified by the respective paper.

⑥ *Coding of API Misuse Causes and Methodologies.* Based on the summary text, we first decided whether the authors of the respective applied *qualitative*, *quantitative*, or *both* methodologies. Then, we conducted open coding [Fli14, p. 373pp] to extract codes describing the API misuse causes and the applied research methodologies independently. Each single code was labeled with a unique identifier to retrieve the publication related to this code. Particularly, `code n:m`, which denotes the `n`th code of the `m`th paper. An example is depicted in Figure 4.2.

⑦ *Categorizing of API Misuse Causes and Methodologies.* Based on codes derived in the previous step, the main author conducted a situation analysis [CFW24, p. 369pp] to summarize, categorize, and interrelate categories. In situation analysis [CFW24], this is achieved by grouping similar codes using so-called *situational maps* and interrelating groups with so-called *relational maps*. We adapted this idea by printing out all codes for API misuse causes and methodologies and clustering them into groups with potential sub-groups, which were labeled accordingly. Afterward, the main author determined potential relations between the main groups and represented them by positioning the group cards as

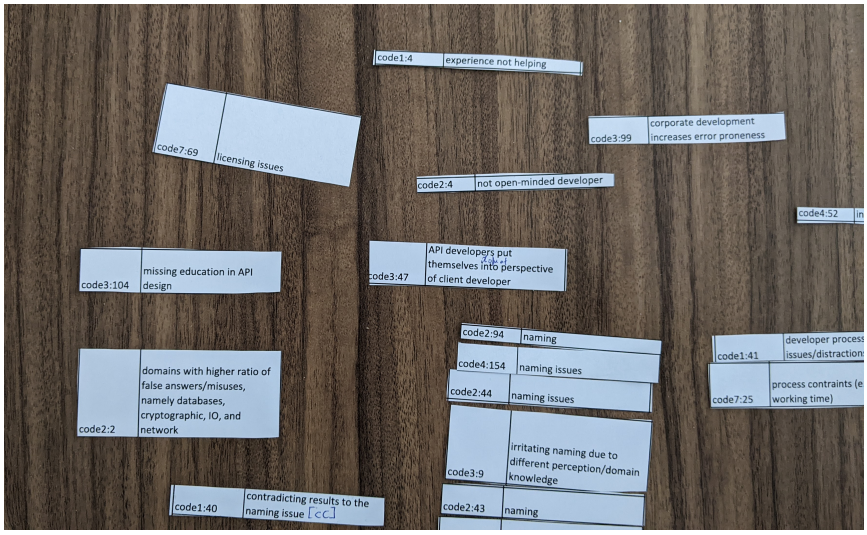
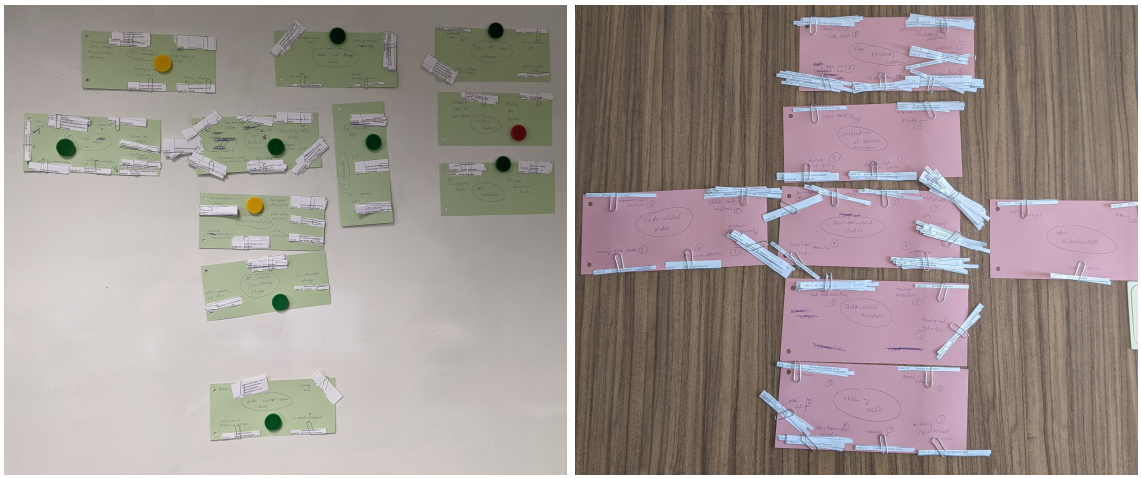


Figure 4.2.: Picture detail of codes derived from the textual summary of API misuse causes.



(a) API misuse causes (b) Methodologies

Figure 4.3.: Result of the categorization of API misuse causes (left, green cards) and methodologies (right, red cards).

depicted in Figure 4.3. After relating the methodologies, we noticed that some codes have been missing in the summary text due to logical reasons (e.g., if data was processed, it is mandatory that it is collected previously). Thus, we re-consulted those papers for which we lacked certain codes and extended the textual summary and the codes accordingly. This re-consultation was done for 61 out of the 65 publications. In a single case, we also corrected a false description and its respective code (i.e., a methodology that was not applied).

Note that coding and categorizing were conducted solely by the main author. While this introduced a subjective bias, the results were discussed with experts from research and industry (i.e., two PhD students, one of them partially employed in a software company, one full professor, and one assistant professor). We detail this discussion on potential threats of this process in Section 4.2.3.

Based on the categorization and the unique identifiers of the codes, we could

(1) trace for each code, in which paper it was mentioned,

(2) relate which categories were discussed in a paper, and

(3) relate API misuse causes with the applied research methodologies (i.e., correlation of methodology and root cause).

Results Meta-Analysis In Table 4.3, we present the list of all selected *relevant* and *partially relevant* papers together with their final decision agreed among the two reviewers and the methodology (i.e., qualitative, quantitative, or both) decided by the main author. Note that the unique identifier `id` refers to all papers considered in the extended set (i.e., in step ④). Thus, `id`s of *non-relevant* papers are excluded from this table. We keep the original `id`s as a reference to our dataset.

We also list the related venues and the publication year to demonstrate the variety of publications among different application domains and analysis time. In detail, our selection contains publications from:

- major software engineering conferences (e.g., ICSE, ASE, ESEC/FSE) and journals (e.g., EMSE, JSS, IST, IEEE Softw.),
- software engineering conferences and journals with a focus on maintenance and evolution (e.g., ICSM/E, SANER), software and empirical analysis (e.g., ISSTA, ESEM), programming language- and software-related (e.g., OOPSLA, COLA), program comprehension (i.e., ICPC), software architecture (i.e., ECSA), computing education (i.e., TOCE),
- conferences with a focus on human-related topics (e.g., CHI, VL/HCC, HCSE, SouthCHI),
- conferences related to a certain application domain, such as security (e.g., CCS, SP, SOUPS, Comp. & Secur.) or quantum computing (i.e., QSW),
- a set of smaller conferences and journals in the software engineering domain.

Considering Figure 4.4, we observed that the selected papers range from 2007 to 2023, while we selected no papers between 2000 and 2006, as well as no papers for the year 2010. For most years, the number of selected papers ranged between one and four, with peaks for 2015 and the years 2018 to 2020 (i.e., between six to eleven publications).

Table 4.3.: List of all relevant and partially relevant publications selected for qualitative analysis with their unique identifiers related to our replication package.

id	Reference	Venue	Year	Decision	Methodology
1	[GWL+19]	COMPSAC	2019	partially relevant	quantitative
2	[ZUR+18]	ICSE	2018	partially relevant	quantitative
3	[GIW+18]	SOUPS	2018	partially relevant	both
4	[OLR+18]	SOUPS	2018	relevant	both
5	[MNY+18]	ICSE	2018	partially relevant	qualitative
6	[NKMB16]	ICSE	2016	relevant	both
7	[LS20]	ICSE	2020	partially relevant	qualitative
9	[ZHKG20]	CHI	2020	relevant	qualitative
11	[PHR19]	SOUPS	2019	partially relevant	both
13	[GALIF20]	CHI	2020	partially relevant	qualitative
14	[Afo15]	Doctoral Thesis	2015	partially relevant	qualitative
15	[NHM+19]	ASE	2019	partially relevant	qualitative
17	[ABF+17]	SP	2017	relevant	quantitative
18	[ABF+16]	SP	2016	partially relevant	quantitative
20	[NDT+17]	CCS	2017	partially relevant	qualitative
24	[Rob09]	IEEE Softw.	2009	partially relevant	both
25	[RD11]	EMSE	2011	partially relevant	both
26	[SC07]	ICSE	2007	partially relevant	qualitative
30	[HL11]	ICPC	2011	relevant	qualitative
32	[ZER11]	WCRE	2011	relevant	quantitative
34	[SM08]	FSE	2008	partially relevant	qualitative
35	[EZG15]	JSS	2015	partially relevant	qualitative
36	[SSD15]	SANER	2015	partially relevant	quantitative
37	[WKA+16]	EMSE	2016	partially relevant	quantitative
38	[SMAR17]	VL/HCC	2017	partially relevant	both
40	[ANBL18]	ICSME	2018	relevant	both
41	[GVIK20]	VL/HCC	2020	partially relevant	qualitative
42	[KMS14]	EMSE	2014	partially relevant	quantitative
43	[DER12]	ICSE	2012	relevant	qualitative
44	[PFM13]	ESEM	2013	partially relevant	qualitative
47	[MKA+18]	VL/HCC	2018	partially relevant	qualitative
48	[DH09b]	ICPC	2009	partially relevant	both
52	[GPT12]	HCSE	2012	partially relevant	qualitative
56	[FHP+13]	CCS	2013	partially relevant	qualitative
69	[MSS18]	JTWC	2018	partially relevant	both
70	[SK12]	ECBS	2012	partially relevant	qualitative
74	[QLL16]	IST	2016	partially relevant	quantitative
75	[SK13]	SouthCHI	2013	partially relevant	qualitative
77	[KFLS18]	JSS	2018	partially relevant	quantitative
78	[MRK13]	ICSM	2013	partially relevant	quantitative
84	[KB23]	COLA	2023	partially relevant	qualitative
88	[TCK21]	TOCE	2021	relevant	qualitative
90	[FWY+19]	JCSU	2019	partially relevant	both
94	[MRARMB+18]	IST	2018	partially relevant	qualitative
95	[WA19]	Comp. & Secur.	2019	relevant	qualitative
99	[CS14]	ICGSE	2014	relevant	quantitative
102	[SK15]	IST	2015	partially relevant	quantitative
104	[RKS+21]	ECSA	2021	partially relevant	qualitative
108	[ARB20]	CLEI	2020	partially relevant	both
110	[UR15]	IEEE Softw.	2015	relevant	both
120	[WLLC20]	ESEC/FSE	2020	relevant	quantitative
121	[CZLF19]	ISSTA	2019	partially relevant	quantitative
123	[HLXX23]	SANER	2023	relevant	quantitative
125	[BCM22]	ICEIS	2022	partially relevant	both
126	[LVBDP+14]	ICPC	2014	partially relevant	quantitative
129	[BVXH20]	EMSE	2020	relevant	both
145	[ZCC+18]	ISSTA	2018	relevant	quantitative
149	[BXHV18]	SANER	2018	relevant	both
151	[BSvdB15]	Softw.Qual. J.	2015	partially relevant	both
153	[JZW+20]	DASFAA	2020	partially relevant	quantitative
154	[BFSK20]	OOPSLA	2020	relevant	both
155	[YHXF22]	IST	2022	relevant	quantitative
157	[LYY+23]	ACSW	2023	partially relevant	qualitative
158	[INPR19]	ESEC/FSE	2019	partially relevant	qualitative
161	[ZWL+23]	QSW	2023	partially relevant	quantitative

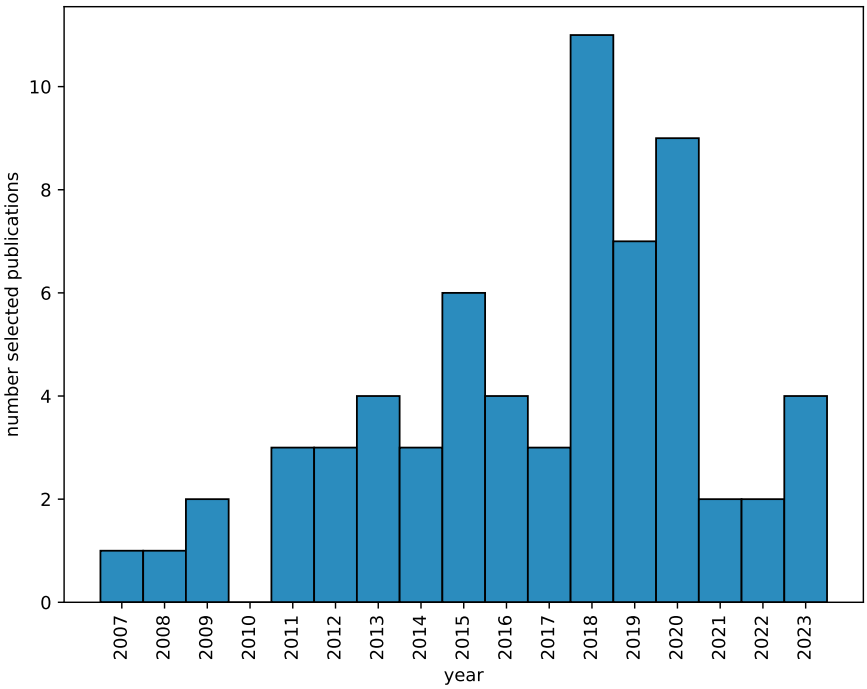


Figure 4.4.: Frequency of Selected Papers per Publication Year

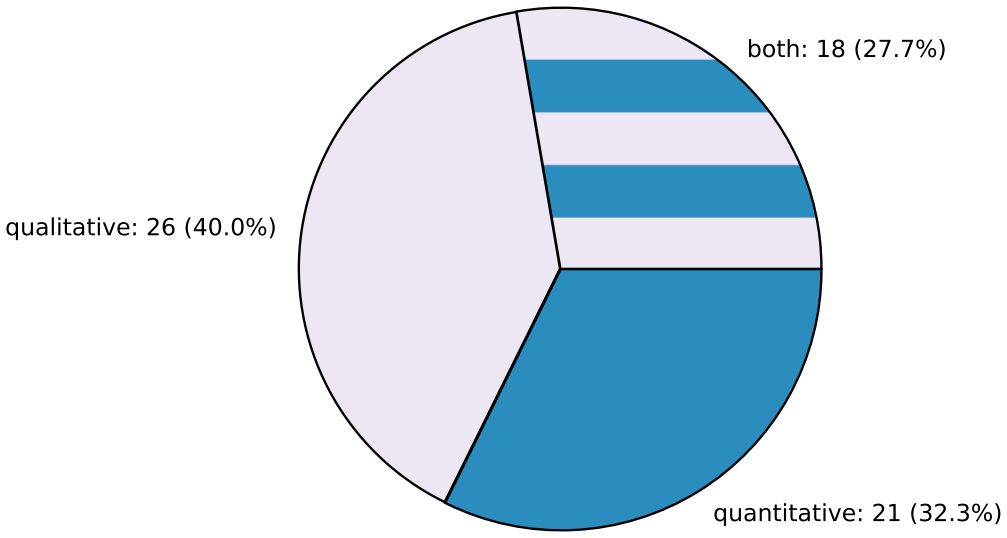


Figure 4.5.: Generally Applied Methodologies

From Figure 4.5, we observed that the proportion of applied methodologies among the selected publications was almost equal, with a slight trend towards more qualitative studies than quantitative ones, namely 40% vs. 32.3%, while 27.7% applied both methodologies.

Therefore, we conclude that our SLR-like selection obtained a sufficiently balanced set of papers regarding applied methodologies, venues, and publication time, while there could be a little bias towards more modern publications (e.g., publication time between 2018 and 2020 vs. 2007 and before) and some domain-specific views (e.g., security-related APIs).

Results API Misuse Causes In Table 4.4, we depict our result on the categorization of step ⑦ for API misuse root causes. We split these into *general root causes* denoted by a single letter and *detailed root causes* as sub-groups of the aforementioned ones represented by a letter-number combination. Note that the order is determined ad-hoc during the coding process and does not represent any severity or prevalence of certain API misuse causes. We shortly describe the codes of the general root causes:

(A) *Complexity and abstraction issues* describe the state of an API in which, due to its design, its API elements are too interrelated to be applicable for client developers or too abstracted to sufficiently understand its interrelations.

(B) *Human API developer issues* describe issues caused by API developers, for instance, due to their education, experience, and/or work processes.

(C) *Human client developer issues* describe issues caused by client developers, for instance, due to their education, experience, and/or work processes.

(D) *API code issues* describe problems according to the implemented API, such as code defects in a library.

(E) *API usability issues* refer to all issues regarding the API implementation, which hardens client developers' usage without representing API code issues.

(F) *False API usage resources* denote issues regarding re-using wrong code from unreliable sources, for instance, suspicious examples from Q&A forums.

(G) *Finding features issues* denote issues regarding the retrieval of functionalities in an API by the client developer.

(H) *API installation issues* describe problems while setting up libraries and APIs, for instance, configuration problems.

(I) *Documentation issues* refer to all kinds of issues related to the API documentation.

(J) *API evolution issues and breaking changes* describe problems related to updates of an API and its library.

(K) *Other context-related issues* refer to all other issues which could not be mapped to any other cause.

For all detailed root causes, we provide an example from the analyzed literature in the appendix (cf. Section A.1.2).

Table 4.4.: Determined codes of API misuse root causes

	general root causes		**detailed root causes**
(A)	complexity and abstraction issues	(A-1)	too complex
		(A-2)	too abstract
		(A-3)	compromise design issues
(B)	human API developer issues	(B-1)	communication issues between API and client developer
		(B-2)	unclear usage scenarios
		(B-3)	missing education on API design
		(B-4)	heterogene API client users
(C)	human client developer issues	(C-1)	developer process-related issues
		(C-2)	missing domain knowledge
		(C-3)	non-helping developer experience
		(C-4)	mindset issues
(D)	API code issues	(D-1)	naming issues
		(D-2)	errors in API
		(D-3)	unclear API error/warning messages
		(D-4)	insufficient defaults
		(D-5)	insufficient initialization of objects
		(D-6)	insufficient error handling
(E)	API usability issues	(E-1)	too many features
		(E-2)	unknown entry points
		(E-3)	API customization issues
		(E-4)	ambiguous/unclear usage
		(E-5)	API incompatibilities
		(E-6)	inconsistent usage compared to similar APIs
		(E-7)	unknown constraints
(F)	false API usage resource	(F-1)	outdated APIs
		(F-2)	using internal APIs
		(F-3)	auto-generation issues
		(F-4)	using insufficient API samples
(G)	finding features issues	(G-1)	findability issues of present API features
		(G-2)	missing API features
(H)	API installation issues	(H-1)	API configuration issues
		(H-2)	technical environment issues
(I)	documentation issues	(I-1)	missing/insufficient documentation issues
		(I-2)	too verbose documentation
		(I-3)	effort to create documentation
		(I-4)	not using documentation
		(I-5)	issues with examples in the documentation
(J)	API evolution issues and breaking changes	(J-1)	inconsistencies due to API changes
		(J-2)	effort supporting old APIs
		(J-3)	non-documented API changes
(K)	other context-related issues	(K-1)	domain
		(K-2)	licensing issues
		(K-3)	API corporate development
		(K-4)	issues due to programming language

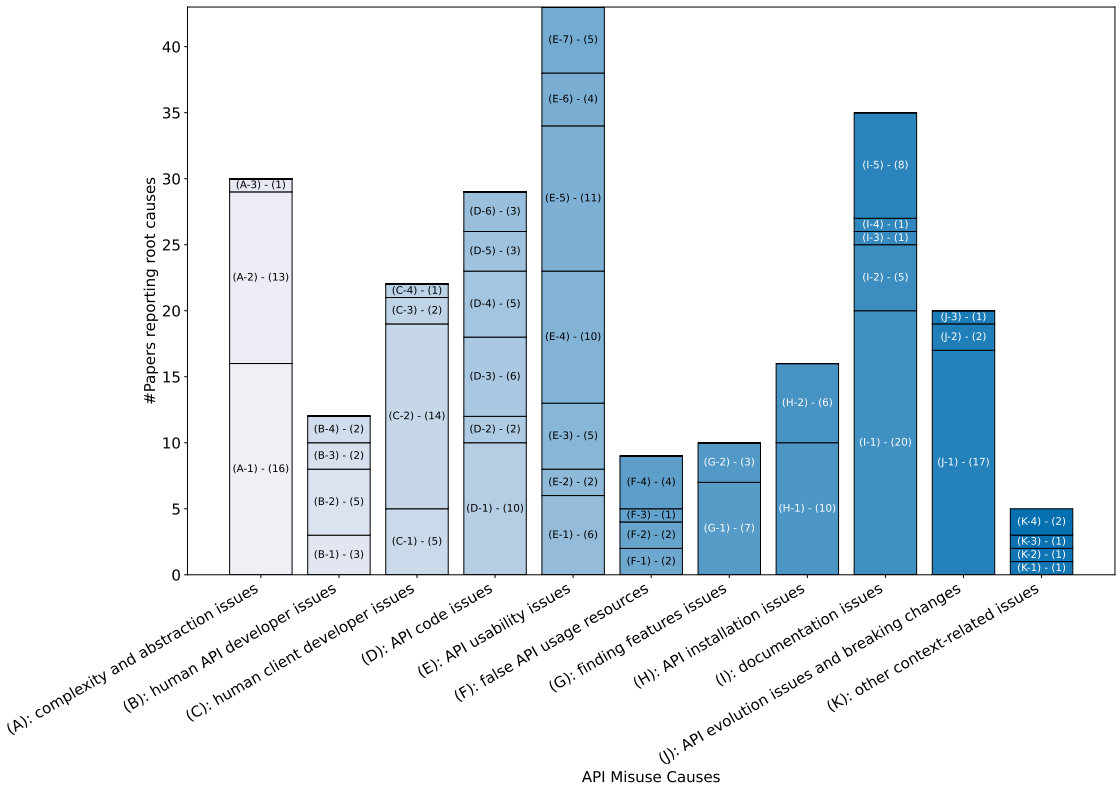


Figure 4.6.: Frequency of discussed API misuse root causes in the selected publications. We depict the detailed root causes since multiple detailed root causes belonging to a general root cause can be present in one paper.

We analyzed how often each detailed root cause was discussed in each paper. This way, we assessed the focus of the research on different API misuse root causes. We depict the absolute frequency in Figure 4.6. Note that in this figure, each *detailed root cause* was counted once for each paper, but it was possible that multiple detailed root causes belong to the same *general root cause* in one paper (e.g., discussion of (E-1) and (E-2) in one paper, while both referred to the general cause (E)).

We observed that most papers found *(E): API usability issues*, followed by *(I): documentation issues*, *(A): complexity and abstraction issues*, and *(D): API code issues* as root causes. In contrast, the least retrieved root causes were *(K): other context-related issues*, *(F): false API usage resources*, and *(G): finding features issues*. It was also interesting to see that fewer papers found the *(B): human API developer issues* compared to the *(C): human client developer issues*. We assume that one reason for this is that a majority of 17 papers considered the client developer perspective, while only eight papers also analyzed the API developer side. Only three publications [Afo15, NDT+17, MKA+18] discussed both sides. In the set of publications, we also found contradictions for two detailed codes on API misuse causes. Namely, Piccioni et al. [PFM13] argued that *insufficient initialization of objects* (i.e., code *(D-5)*), for instance, a constructor not completely setting up an object, could not be confirmed. In another study, Aghajani et al. [ANBL18] found contradicting results regarding the negative impact of naming variables and functions (i.e., code *(D-1)*).

A detailed discussion of the detailed root causes can be found in the appendix (cf. Section A.1.1).

Insight C&P-1 (RQ C&P-C): Diverse Root Causes

Based on the scientific literature, API misuses have multiple and very diverse root causes, with eleven general root causes and a further 44 detailed root causes as well as causes from different perspectives, namely, developer-, process-, and technical-related causes. The most discussed general root causes refer to issues regarding API usability, documentation, API complexity and abstraction, and code issues.

Results Interdependent API Misuse Causes Despite this list, the general and detailed root causes cannot always be viewed as independent of each other. For instance, insufficient API documentation (i.e., code (I-1)) can cause configuration issues (i.e., code (H-1)). In another case, communication issues between API and client developer (i.e., code (B-1)) can be caused by a missing communication means, namely, the documentation (i.e., code (I-1)). These examples illustrate the complexity of the interrelations of the root causes of API misuses.

Since this was – to the best of our knowledge – the first attempt to summarize this variety of root causes of API misuses, their interrelations were also not discussed previously. However, these interrelations are important to understand the causes and to effectively derive prevention mechanisms since intervention on one root cause can interfere with another one. Therefore, we developed a process to derive hypotheses of interrelations – so-called *views* – as a basis for further empirical research on API misuse root causes.

In detail, we summarized multiple general root causes to logically coherent and labeled groups, while these groups were not forced to be disjoint. These group labels represent a *perspective*. Then, the main author argued based on his own experience from reading related studies and logical relationships of the detailed root causes, possible interrelations. This discussion is presented in Table 4.5 Note that at the current point-in-time, we are not

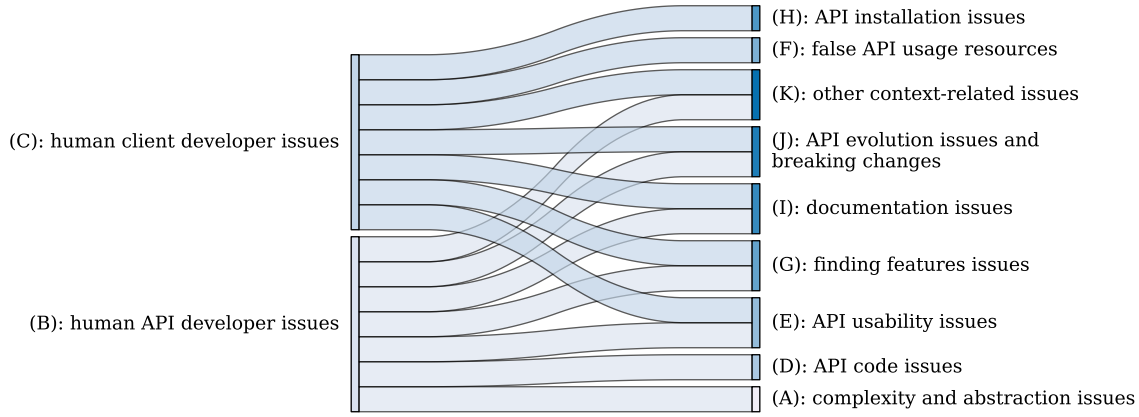


Figure 4.7.: Developer-perspective view on API misuse root causes as Sankey diagram with the cause-effect chain flow from left to right.

aware of any empirical work nor did we collected empirical evidence regarding interrelations. This way our views are limited in their expressiveness. However, this procedure only serves to formulate hypotheses, which can be falsified or improved in subsequent research.

A view represents a simplified cause-effect chain, by which *main root causes* are selected as origin based on a specific *perspective*, and we assess their *influence on all other root causes*. Based on our list of general root causes (i.e., **(A)-(K)**), we suggest the following three perspectives:

- ***developer-perspective view*** with the main root causes *(B) human API developer issues* and *(C) human client developer issues*
- ***process-perspective view*** with the main root causes *(F) false API usage resource*, *(I) documentation issues*, *(J) API evolution issues and breaking changes*, and *(K) other context-related issues*
- ***technical-perspective view*** with the main root causes *(A) complexity and abstraction issues*, *(D) API code issues*, *(E) API usability issues*, (H) API installation issues, *(J) API evolution issues and breaking changes*, and *(K) other context-related issues*

Thus, we distinguished the main root causes between the developers (e.g., API or client developer), the development process, and the technical aspects during API usage. Note that *(J) API evolution issues and breaking changes* and *(K) other context-related issues* are the main root causes for *process-perspective* and *technical-perspective views*. *(J)* is a root cause because of the detailed causes *(J-2) no support for older APIs* and *(J-3) non-documented changes*, which relate to the process-perspective view, while *(J-1) inconsistencies through API changes* relates more to the technical-perspective view. For *(K)*, the detailed cause *(K-4) issues with the programming language* maps to the technical view, while the others (i.e., *domain (K-1)*, *licenses (K-2)*, and *corporate development (K-3)*) relate to the process-perspective view.

We present the constructed views as Sankey diagrams in Figure 4.7 (i.e., developer-perspective), Figure 4.8 (i.e., process-perspective), and Figure 4.9 (i.e., technical-perspective). The main root causes are on the left-hand side and are connected to their influenced cause on the right-hand side by colored bars. Typically, Sankey diagrams represent

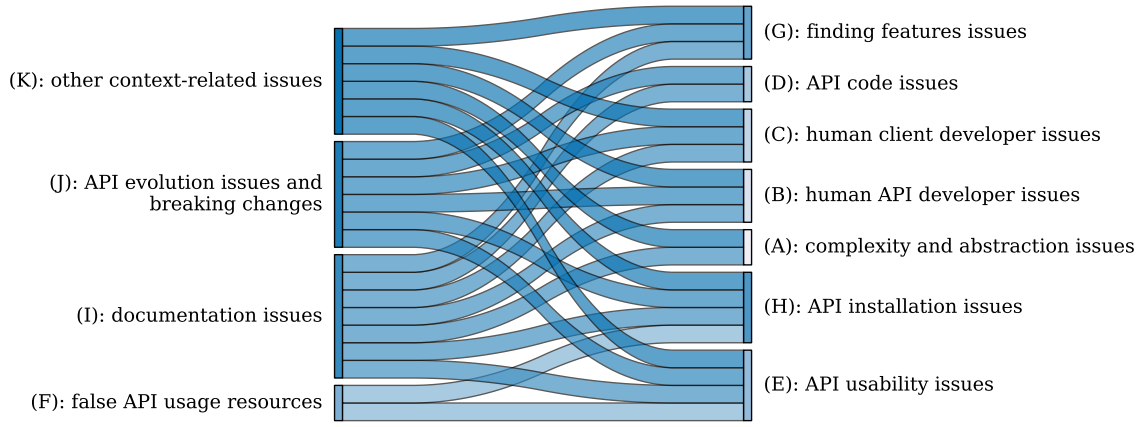


Figure 4.8.: Process-perspective view on API misuse root causes as Sankey diagram with the cause-effect chain flow from left to right.

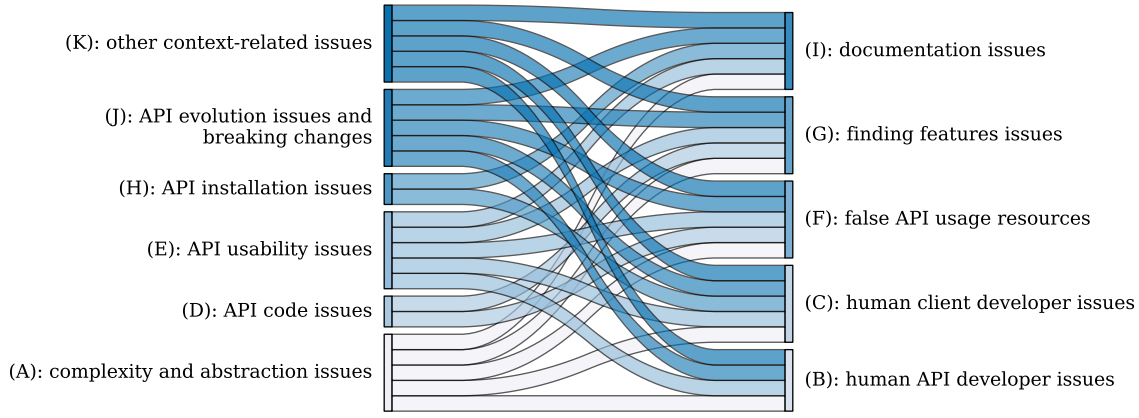


Figure 4.9.: Technical-perspective view on API misuse root causes as Sankey diagram with the cause-effect chain flow from left to right.

Table 4.5.: Justification of three possible *views* on the interrelations on API misuses root causes

View	Justification
developer-perspective (Figure 4.7)	In this perspective, we assessed issues by API developers *(B)* as a single cause for *(A) complexity and abstraction issues* and *(D) API code issues*. These issues were mainly caused by API developers influence the API design and thus complexity and abstraction as well as the API implementation, and thus its code. Similarly, we decided that *(F) false API usage resources* and *(H) API installation issues* are solely (i.e., in this perspective) caused by client developers *(C)* since installation and configuration, as well as reused code samples of an API, occur on the client side. Within this perspective, all other causes are influenced by both kinds of developers *(B)+(C)*, however, with different proportions. On the one hand, API developers *(E)* have slightly more influence on the documentation *(I)*, the findability of API features *(G)*, and the API evolution *(J)*. On the other hand, client developers' *(C)* issues are assessed to have more effect on the API usability *(E)* and other related contexts *(K)*. Nevertheless, the issues of API usability *(E)* cannot be clearly assigned to one kind of developer. While API developers definitely influence the number of features *(E-1)*, potential incompatibilities *(E-5)*, and inconsistent usage compared to similar APIs *(E-6)*, client developers, due to their lack of knowledge on an API do not know entry points *(E-2)*, have customization issues *(E-3)*, misunderstand the usage *(E-4)*, and do not know constraints of that usage *(E-7)*.
process-perspective (Figure 4.8)	From a development process perspective, we assessed no mono-causal effects. In contrast, we denoted the API usability *(E)* and API installation *(H)* issues caused by all four main root causes that are process-related. The effects, however, are of a different nature. Particularly, false API samples *(F)* and breaking changes *(J)* lead to code issues, causing problems in the usability *(E)* or the installation *(H)*. A misleading documentation *(I)* or a complex domain *(K)*, on the other hand, may trigger issues on the client side during configuration *(H)* or during customization *(E)*. For complexity *(A)*, we only assess insufficient explanations from the documentation *(I)* and a too-complex domain *(K)* as an influence. Insufficient documentation (I), breaking API changes *(J)*, and the domain of the API *(K)* influence the communication and the understanding of API- *(B)* and client-developers *(C)*, as well as causing issues on the findability of API features *(G)* due to non-documented updates, a less communicative domain, or the complexity of the features, which prohibit efficient search. API code issues *(D)* can be caused by the documentation *(I)*, for instance, if it contradicts the code and thus causes naming issues or unclear error messages, as well as by the API updates, which can cause all kinds of code issues.
technical-perspective (Figure 4.9)	Within the technical perspective, we argued that coding issues of the API only influence false resource API usage *(F)*, for instance, if the code unintentionally allows access to internal APIs as well as findability issues *(G)* in case naming issues hardens finding proper API elements. Installation issues *(H)* can cause documentation *(I)* to be too verbose or too abstracted in case too many installation or configuration possibilities exist. It also can cause problems on the client side*(C)*, for instance, too long setup processes or a negative mindset due to a complicated installation. Complex APIs *(A)* can cause an increased number of client users (i.e., more applicable functionality), which API developers have to be aware of *(B)*, a higher chance of missing domain knowledge on the client side *(C)*, more complex search *(F)*, and a more verbose documentation *(I)*. Usability issues *(E)* can complicate the communication of API developers towards their client users *(B)*, increase the effort of using APIs, and thus have a negative effect on client developers' processes *(C)*, causing the usage of insufficient examples due to unknown entry points *(F)*, and negatively impact findability *(G)* or result in too verbose documentation *(I)* due to too many features. API evolution issues *(J)* can cause issues when changes are not properly communicated by API developers *(B)* and if the client developers are not familiar with the evolution strategy *(C)*. Having multiple versions increases the chance of using outdated examples *(F)* or finding many different variants of possible usages *(G)*. Finally, the documentation may lack sufficient information on the new features *(I)*. Context-related issues such as the programming language *(K)* cause issues for API and client developers alike due to communication issues through the language *(B)* as well as the not-helping experience from other programming paradigms (e.g., object-oriented vs. functional). Non-popular programming languages may have issues with non-reliable auto-generated code *(F)*. In the case of a more verbose programming language, findability can be complicated *(G)* or the documentation can be too verbose as well *(I)*.

the influence of causes by the thickness of these bars. However, currently, we cannot provide any quantitative justification for such influences.

By these views, we can formulate hypotheses of the interrelations among API misuse root causes, such as *client developer issues cause the usage of false API resources*. This way, future experiments can analyze particular this hypothesis in a developer experiment or ask client developers in an interview regarding certain interrelations to manifest or falsify the hypothesized interrelations (e.g., client developers cause problematic resource usage). Especially quantitative studies can refine the proportion of influence (i.e., the thickness of the Sankey bars).

Moreover, our views depict suggestions for possible interrelations. In case those are found to be invalid or non-sufficient (e.g., the existence of further views), other researchers can alter or add additional views.

Insight C&P-2 (RQ C&P-C): Root Cause Relations by Views

Based on the set of API misuse root causes obtained from the literature, we provide a mechanism to derive hypotheses of the interrelations of the root causes as so-called ***views****. Moreover, we provide and justify three views, namely from the* ***developer-perspective****, the* ***process-perspective****, and the* ***technical-perspective view****.*

Results Methodologies of API Misuse Causes Studies Similar to the codes of the root causes, we summarize in Figure 4.10 the codes obtained for the research methodologies of those publications using general codes with **(A)-(G)** and sub-codes with a letter-number reference. The obtained steps follow a typical empirical research approach [RAB+20] with an initial *(A) raw resource collection*, namely, the process of collecting data on which the analysis takes place. This collection is sometimes aligned with a *(B) preparation of resources and studies*-step, in which data is sampled, selected, filtered, and studies are set up. Afterward, empirical studies are conducted, which we categorize as *(C) code-related studies* (i.e., both API and client code), *(D) developer-related studies* (i.e., on both API and client developers), or *(E) other studies/analyses* (i.e., API-related documents such as documentation). Depending on the kind of study, different *(F) study-related processes* are applied, such as recording steps. Finally, the *(G) obtaining results* step describes how the output from the studies is transformed into research findings (i.e., determined API misuse root causes). For all sub-codes on the methodology, we provide an example from the analyzed literature in the appendix (cf. Section A.1.3).

Note that the repeated analysis of the methodology discussed in step ⑦ concentrated on the codes for data collection (i.e., (A)), for the kind of study (i.e.,(C), (D), (E)), and for the step of obtaining the study results (i.e., (G)).

Insight C&P-3 (RQ C&P-C): Typical Empirical Study Structure

The methodologies used to analyze API misuse root cases follow a typical empirical approach of resource collection, preparation, conducting studies combined with study-supporting steps, and a final inference step of the root causes. Regarding the kinds of studies we found code-related, developer-related, and other studies (e.g., document-analysis such as documentation or forum discussions).

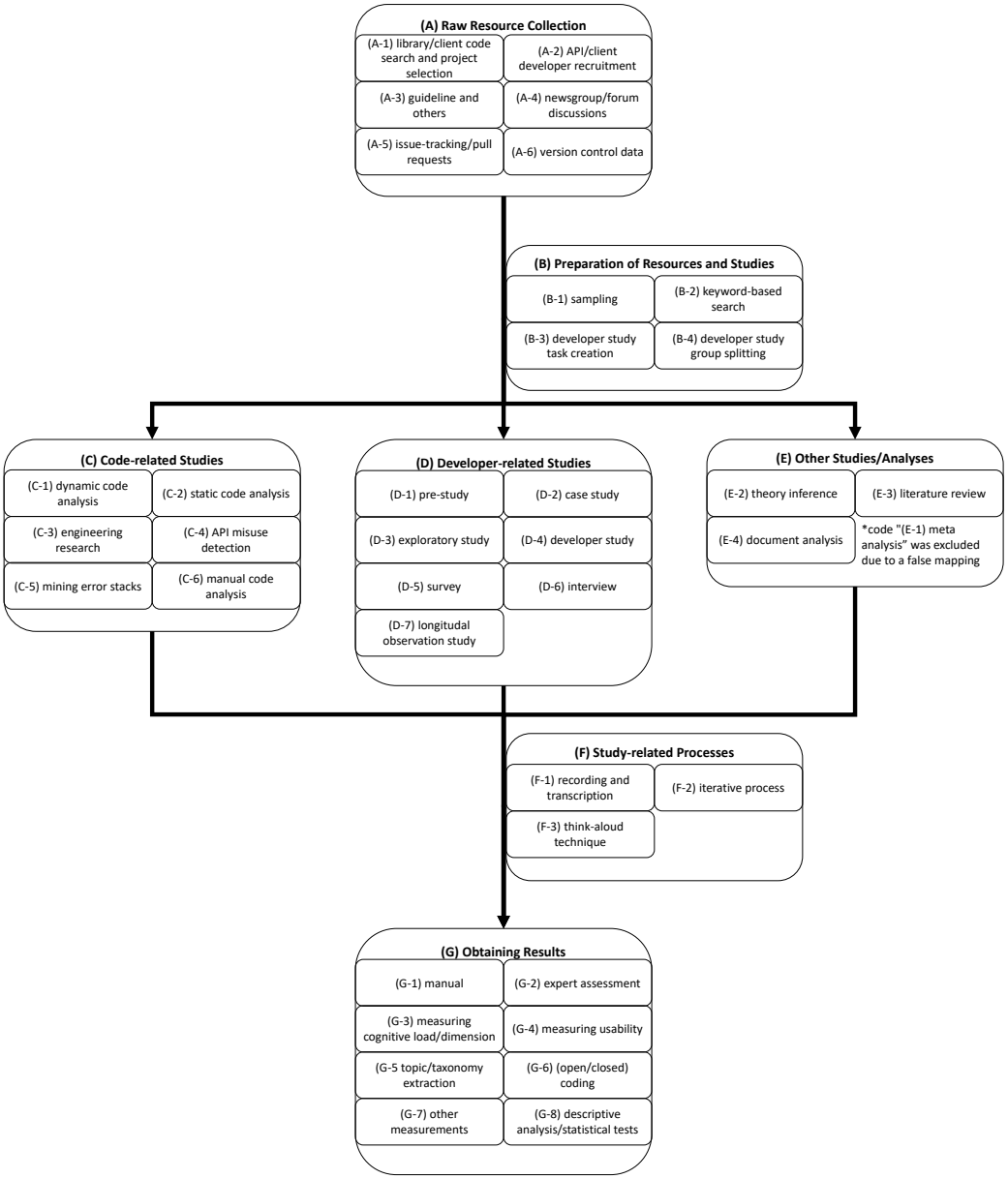


Figure 4.10.: Overview and relation of determined methodology codes of API misuse root cause analysis

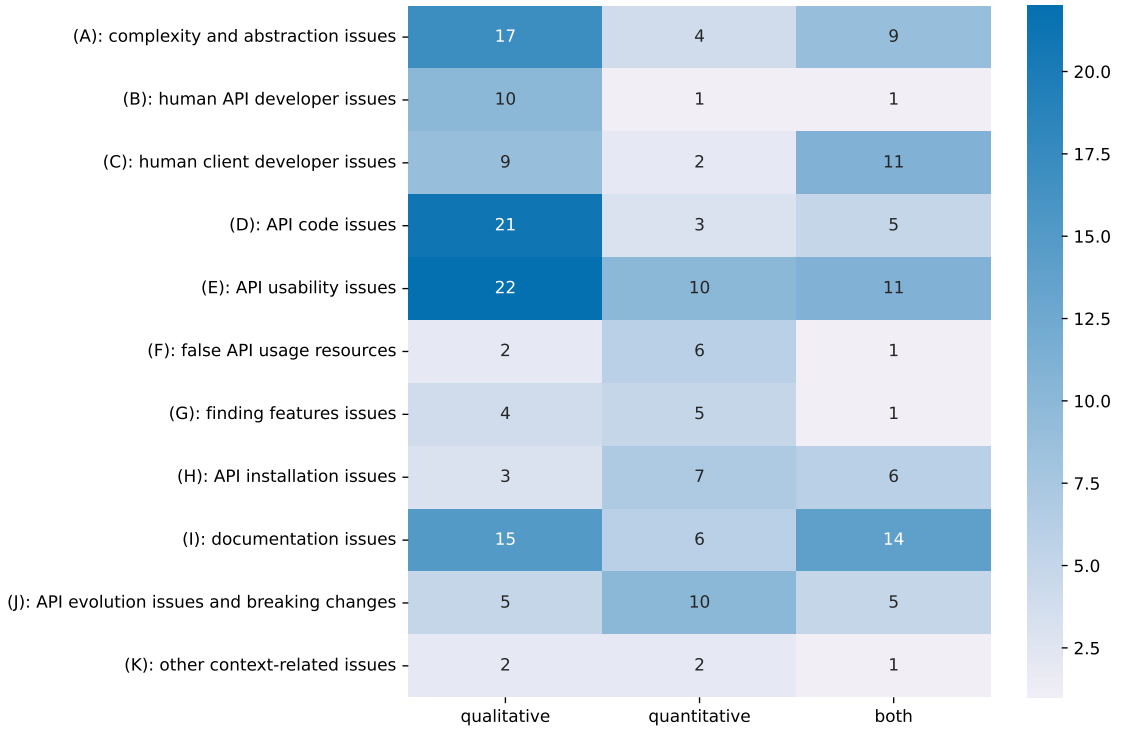


Figure 4.11.: Correlations of general root causes and qualitative/quantitative methods as heat map.

Results Mapping of API Misuse Causes and Research Methodologies Due to the mapping of codes of root causes and methodologies to their respective papers, we were able to interconnect which methodologies were correlated with which observed root causes[4]. Note that *mapping a methodology to a root cause does not necessarily mean that this root cause is obtained by this method.* It only means that this root cause is also found in a paper using this specific methodology. Thus, this mapping allows an indication and not necessarily an implication of applied methods. However, we could identify potential research gaps (i.e., which methodologies have not been applied in root causes analysis).

Using the manual decision on methodologies for the publications (i.e., *qualitative*, *quantitative*, or *both*), we depict in Figure 4.11 a heat map representing which root causes are correlated with which methodology. We found a larger correlation of *qualitative* studies detecting the root causes *(A) complexity and abstraction*, *(B) human API developer*, *(D) API code issues*, *(E) API usability issues*, and *(I) documentation*. Note that except for *(B) human API developer* and *(D) API code issues*, all other root causes were also correlated with nine or more studies using *both* methodologies. For *(F) false API usage resources*, *(G) finding features issues*, *(H) API installation*, and *(J) API evolution issues and breaking changes*, more *quantitative* studies were mapped. We found for the root cause *(K) other context-related issues* the least number of studies, which were equally distributed among the different methodologies. Thus, the most frequently found root causes (i.e., root cause

[4] A similar mapping technique has been also conducted by Cummaudo et al. [CVG19], however, they mapped study techniques and research types

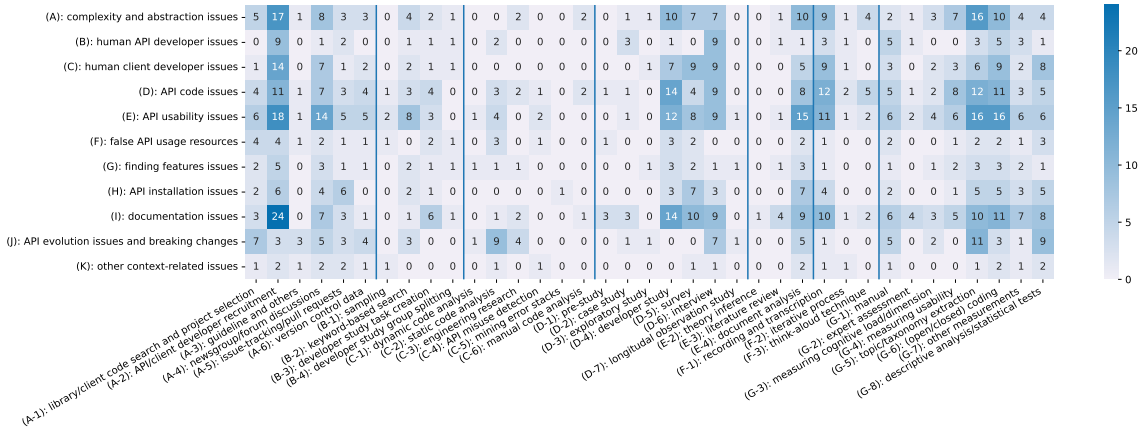


Figure 4.12.: Correlation of general root causes and applied research methodologies as heat map.

codes (A), (D), (E), (I)) were mostly mapped to qualitative studies.

By using the codes of methodologies (cf. Figure 4.10), we also correlated the more detailed methods with the general root causes depicted as a heat map in Figure 4.12. Note that we refer to codes representing root causes as *root cause code* and those representing a methodology as *method code*. We also provide a more detailed mapping of the detailed root causes in the appendix (cf. Section A.1.4).

Method Code (A): We observed that many root causes are based on recruiting API and client developers (i.e., method code (A-2)) as well as newsgroup and forum discussions (i.e., method code (A-4)).

Method Code (B): The *preparation of resources and studies* had only a few heat points, namely using keyword-based search (i.e., method code (B-2)) to find documents for analysis of the root causes *(A) complexity and abstraction* and *(E) API usability*.

Method Codes (C), (D), and (E): We found surprisingly few code-related studies (i.e., method code (C)) with a single 'hot spot', where the root cause *(J) API evolution issues* were found using static code analysis (i.e., method code (C-2)). We found many more developer studies (i.e., method code (D)) and other studies/analyses (i.e., method code (E)) investigating various root causes. For the root causes *(A) complexity and abstraction*, *(C) human client developer*, *(E) API usability issues*, and *(I) documentation*, we found developer studies such as coding experiments (i.e., method code (D-4)), surveys (i.e., method code (D-5)), and interviews (i.e., method code (D-6)) among developers. For those root causes, we also found many studies analyzing documents (i.e., method code (E-4)) such as online forum discussions. Regarding the root causes *(D) API code issues*, the above was also true except for fewer surveys (i.e., method code (D-5)).

Another interesting result was the root cause of *(J) API evolution issues and breaking changes*, which was found by a set of three different study types, namely static code analysis (i.e., method code (C-2)), interviews (i.e., method code (D-6)), and document analysis (i.e., method code (E-4)). In our perspective, finding this root cause with different methods strengthens its validity.

Method Code (F): For the aspect of study-related processes (i.e., method code (F)), many recording and transcription techniques (i.e., method code (F-1)) were conducted. Particularly these technique appeared in developer experiments or in-person interviews.

Method Code (G): Finally, in the obtaining results step (i.e., method code (G)), we found many techniques related to qualitative methods, namely topic/taxonomy extraction (i.e., method code (G-5)) and open/closed coding (i.e., method code (G-6)). On the contrary, we found fewer techniques related to quantitative methods, namely measuring cognitive load/dimension (i.e., method code (G-3)) or descriptive analysis/statistical tests (i.e., method code (G-7)).

These results draw the image that, based on our mapping, there exist far fewer quantitative methodologies than quantitative ones. While qualitative methods enable researchers to explore the domain of API misuse root causes, it does not give evidence on prevalence and severity. Thus, we suggest that based on the set of known root causes, future research should concentrate on quantitative methods.

We also observed that a majority of root causes were obtained based on documents or developers. Particularly, surveys and interviews may have issues of developers' bias towards certain root causes. While this is partially compensated with a larger set of developer studies, we also suggest concentrating on more code-related studies. In this aspect, the main root causes from the *technical-perspective view* (cf. Figure 4.9) are interesting. For instance, we mapped no study using error stack mining (i.e., method code (C-5)) for the root cause *(D) API code issues* or no dynamic or static code analysis (i.e., method codes (C-1) and (C-2)) for the root cause *(H) API installation issues*.

Note that not all root causes can be analyzed using all kinds of studies, such as code analysis (i.e., method codes (C-1) and (C-2)) for root causes *(B) human API and (C) client developer*[5]. However, we see the potential of developer studies (D-4) analyzing the effect of the root cause *(B) human API developer issues*.

Insight C&P-4 (RQ C&P-C): More Qualitative Studies

We found that the most frequently mentioned API misuse root causes are typically correlated with more qualitative than quantitative methods. In more detail, we observed that those root causes were merely found by developer-related studies ranging from surveys and interviews up to developer studies, such as coding experiments as well as other studies such as document analysis (e.g., of online forum discussions).

Insight C&P-5 (RQ C&P-C): More Diverse Studies for API Evolution Issues

We found that only the root cause of API evolution issues and breaking changes *was frequently correlated with all three kinds of studies, namely, code- and developer-related studies, as well as other studies, such as on related documents. In detail, we found few code-related studies for most discussed root causes and even fewer coding experiments for issues caused by API developers.*

Implications Our results represent a first overview of known root causes of API misuses as well as the frequency by which they are present in the scientific literature (i.e., **Insight**

[5]Recall that the two mapped studies using static code analysis to the root cause of human API developer issues are just a result of correlation. Both related papers, namely, did not find this root cause using static code analysis.

C&P-1). In this form, our results inform practitioners *which* root causes exist (i.e., **Insight C&P-1**). Currently, we *neither know how these root causes interrelate nor can we quantify their prevalence and severity on API misuses* (i.e., **Insight C&P-2** and **Insight C&P-5**).

For the research community, these results allow us to identify potential gaps in the scientific literature on API misuse root causes (i.e., **Insight C&P-4** and **Insight C&P-5**) as well as plan and conduct studies to close these knowledge gaps (i.e., **Insight C&P-2** and **Insight C&P-3**).

In detail, we

- offer the first overview of API misuse root causes (i.e., **Insight C&P-1**);
- provide a *methodology to hypothesize and plan further research on interrelations of API misuse root causes*, which, due to their lack of knowledge, can impede effective prevention mechanisms (i.e., **Insight C&P-2**);
- provide the typical structure of research methodologies in research of API misuse root causes that helps to plan further studies or to apply new methodologies, which were not used previously (i.e., **Insight C&P-3**);
- identify that most root causes originate from qualitative studies, which provide a good exploration of the topic of API misuse root causes, however, *cannot answer questions on the prevalence and the severity of API misuses in practice* (i.e., **Insight C&P-4**);
- found potential gaps in the kinds of studies of root causes, which can be the basis for more in-depth analysis with known or different kind of studies and research methodologies (i.e., **Insight C&P-5**)

4.2.3. Threats to Validity

We shortly discuss potential threats to *internal* and *external* validity regarding our insights on **RQ C&P-C**.

Internal Validity We only applied the single scholar search engine Google Scholar, which might introduced a bias towards selected papers regarding the used keywords and search time [KLvN+20]. Even though we applied different techniques (i.e., keyword search as well as forward and backward snowballing), we might miss relevant papers dealing with API misuse root causes. A further selection bias could be introduced by the reviewers due to subjective assessment. For instance, the reviewers efficiently decided on the relevance using the paper's title as a first indicator, which could be misleading.

While the selection of papers was agreed upon the two reviewers, the coding step and the decision was solely conducted by the main author, which introduced a clear bias. While it would have been too time-consuming for another researcher to re-validate the textual summaries, codes, and final decisions, we specifically documented all steps and published all results[6] to allow transparency and potential independent replication.

The frequency of root causes could be influenced by the construction and separation of inferred root causes itself. For instance, the root of (K) other context-related issues is likely to be underrepresented in research since it depicts causes which could not be mapped

[6] http://doi.org/10.5281/zenodo.15594600

to other causes. Similarly, the frequency of other root causes could be larger or lower depending on the categorization.

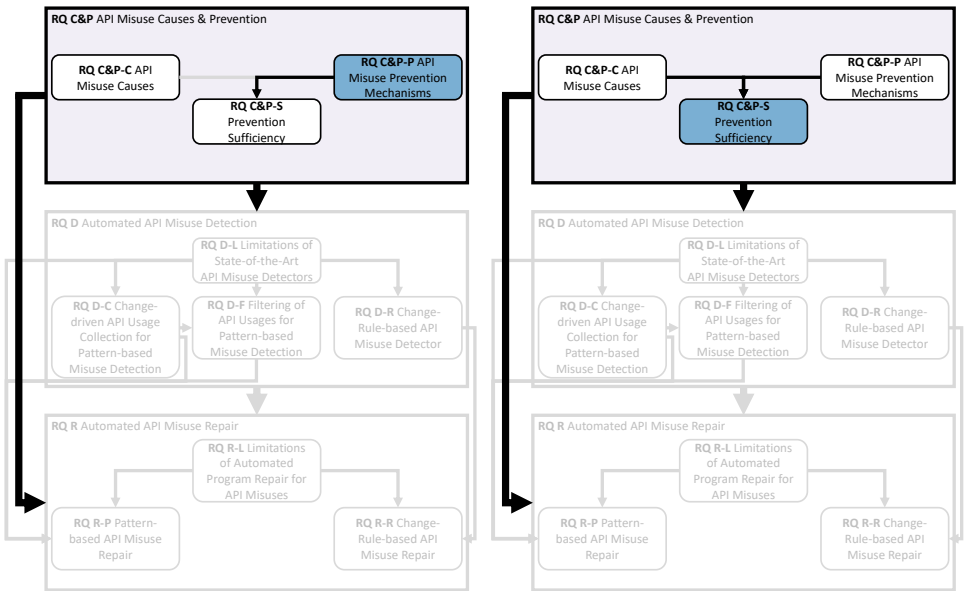
When mapping codes of research methodologies and root causes, this only gave restricted insights on whether the related methodologies found these root causes. While we assumed this for the 'hotter' spots within our heat maps (cf. Figures 4.11 and 4.12), these correlations might be a result of randomness. We suggest, for a conclusive decision on which methodology is applied, to consult the respective publication by using the codes used in our dataset.

External Validity The presented set of root causes only relies on the relevant scientific literature and, thus, does not necessarily replicate the complete set of root causes. In detail, the reported API misuse causes only depict the state of the knowledge in the scientific literature at the time of analysis. Further empirical studies might identify further root causes or may falsify present root causes in our obtained list.

As reported in our selected set of papers, we found more modern publications and a flavor of security-related literature. Thus, the selected API misuse root causes can be influenced by the selected time frame and the considered domain. We suggest that domains that are not explicitly or only partially represented by the selected publications re-validate whether the found API misuse root causes are applicable. Moreover, based on our data, we cannot derive the effect of whether root causes change over time.

We only provide *hypotheses* of interrelations of API root causes as *views*. We state that these views do not necessarily represent real interrelations but only our subjective perspective, and thus require further scientific evidence, for instance, by empirical studies.

4.3. API Misuse Cause Prevention



In this section, we analyze prevention strategies and techniques targeting the API misuse root causes, and thus, we answer the research question **RQ C&P-P**. Once again, we apply an SLR process as suggested by Kitchenham and Charters [KC07], namely, *planning*, *conducting*, and *reporting*. For *planning*, we already defined the research questions (i.e.,

Table 4.6.: Overview of related systematic literature analyses on API misuse prevention compared to ours

Reference	#Publications	Time Frame	Discussed Causes	Prevention Mechanism
[LGS21]	369	1994-2020	API evolution	automated support
[NAP18]	36	2000-2016	API documentation	automated support
[CVG19]	21	NA-2019		recommendation
[Zib08]	NA	NA	API usability	recommendations
[BFHM12]	28	2004-2011		recommendations
[RBK+13]	67	2000-2010		automated support
[RTP19]	47	1998-2018		both
[PDHR23]	65	1974-2021		both
[McG23, p. 13pp, p. 22pp]	73	NA-2023	client developer, API usability, API documentation	recommendations
[RRS23]	109	2016-2022	client developer	recommendation
[AAWX23]	68	2014-2022	finding features	automated support
Ours	**411**	**2000-2025**	**general**	**both**

RQ C&P-P) and discussed the necessity of this SLR in the light of related surveys (cf. Section 4.3.1). For the *conducting* phase, we discuss the methodology of our review process in Section 4.3.2. Subsequently, we *report* the results based on prevention mechanisms based on quantitative and qualitative analysis and determine their state of scientific evidence. Moreover, based on these results and the determined research gaps, subsequently, we discuss future research directions for targeting API misuse root causes and thus target **RQ C&P-S** in Section 4.3.3.

4.3.1. State-of-the-Art of Meta-Analysis of API Misuse Prevention

First, we consulted previous literature reviews related to the topic of prevention of API misuses in comparison to our review-like study depicted in Table 4.6. For that purpose, we read the reviews already known from misuses causes (i.e., in Table 4.1) and found all but one (i.e., Bonorden and Riebsch [BR22]) discussing either *recommendations* or *automated support* as prevention mechanisms. In addition, we found three further reviews due to our own literature review process (cf. presented in Section 4.3.2), namely the surveys by Robillard et al. [RBK+13], by Ryan et al. [RRS23], and by Alhosaini et al. [AAWX23], all of them concerned with API misuse prevention but not with API misuse causes.

Based on Table 4.6, our review targeted all API misuse causes as identified in the previous section. Moreover, we discussed both kinds of prevention mechanisms for API misuse root causes, which was only true for the surveys by [RTP19] and [PDHR23]. Our study contained publications from the most recent time frame, including the years 2024 and 2025. Moreover, we obtained the largest number of publication in this set with 411 publications.

Subsequently, we summarize the the single reviews per discussed root cause.

API evolution: In detail, Lamothe et al.[LGS21] discussed automated support for targeting API evolution issues, such as techniques supporting the organization of changes, automated migration tools, or automated updating of the documentation.

API documentation: Nybom et al. [NAP18] discussed automated support techniques, such as generation and maintenance support, while Cummaudo et al. [CVG19] summarized recommendation as a guideline discussing different dimensions of guidelines.

API usability: Many related surveys reviewed prevention mechanisms for API usability. Zibran [Zib08] provided 22 recommendations, and Burns et al. [BFHM12] focused on recommendations for the design, documentation, and methodology. Robillard et al. [RBK+13] focused on automated support of API specification inference techniques, which, for instance, can be applied to automated code suggestions. In another survey, Rauf et al. [RTP19] included automated support, such as API usability assessment tools, but also recommendations, for instance, including usability studies in early development processes of APIs. Instead, Patnaik et al. [PDHR23] focused on the usability of security APIs and considered automated support like security assessment tools and recommendations, such as principles of secure software development of APIs.

Other Causes: Ryan et al. [RRS23] presented potentially false assumptions on secure development and thus reviewed recommendations targeting the root cause of *client developers*. Alhosaini et al. [AAWX23] analyzed automated support for the root cause of *finding API features*, namely recommendation systems. McGregor [McG23] discussed several recommendations for the root cause of client developers (e.g., better teaching environments), API usability (e.g., providing test modes), and API documentation (e.g., including more code examples).

Note that some surveys also provided prevention mechanisms, which were applicable to other root causes as well. However, none of them could consider the large perspective on prevention mechanisms targeting all 11 root causes as discussed previously, since these were previously not apparent. This way, our survey provides a benefit to the state of the research.

4.3.2. Meta-Analysis API Misuse Prevention for Misuse Causes

Methodology We obtained the publications on API prevention mechanisms using the process depicted in Figure 4.13. We started based on the 66 publications found for API misuse causes (i.e., including the one identified as duplicate in Figure 4.1) in step ①. We conjectured that publications discussing root causes of API misuses also likely discuss potential prevention mechanisms or refer to them. Then, we conducted a *single step forward and backward snowballing* (②). In detail, we applied ordinary snowballing [Woh14] but only considered those publications cited by at least one of the papers within our initial set (i.e., backward snowballing) or that cited at least one of the papers of this initial set (i.e., forward snowballing). We did not consider publications going beyond this (e.g., publications citing those papers found by backward snowballing) since we found the time frame and number of retrieved publications sufficient for our analysis. Regarding the forward snowballing (③), we leveraged the '*cited by*'-functionality by Google Scholar[7]. We conducted the forward snowballing in a time frame from November 11th, 2024, to January 8th, 2025. This time frame is important since search results can differ over time [KLvN+20]. For backward snowballing (④), we consulted the respective reference list of the paper within our initial set and consulted the publications in the time frame from January 8th to January 15th, 2025. For both variants of snowballing, we applied the following criteria for initial selection:

[7] cf. https://scholar.google.com/ last accessed: 2025/03/20

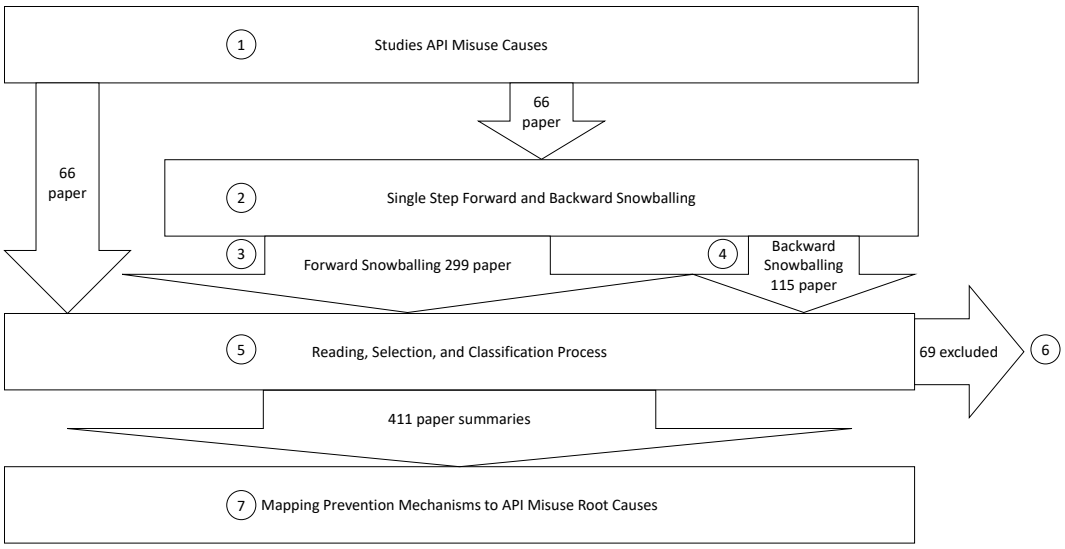


Figure 4.13.: Overview of SLR-like process for meta analysis of API misuse prevention

- the publication is a peer-reviewed article;
- it is written in English;
- it is published between the years 2000-2025;
- the title and additionally the abstract indicate that a paper considers a prevention mechanism (i.e., recommendations or automated support) targeting root causes of API misuses.

This way, the main author found 299 additional publications via forward snowballing (i.e., ③) and 115 publications using backward snowballing (i.e., ④). Including those 66 publications, we obtained a set of 480 publications as input of the *reading, selection, and classification process* in step ⑤.

Reading: First, the main author processed each paper using the following procedure:

1) read the title and the abstract
2) read the conclusion and skim through the discussion or implication section, if applicable
3) skimmed through the evaluation and results section, if applicable
4) skimmed through the lessons learned and recommendation section, if applicable

For replicability, the main author summarized the prevention mechanisms of each paper in short textual form.

Selection: While reading, we excluded those publications (i.e., in addition to the decision criteria discussed before) that either did not suggest a prevention mechanism or that essentially represented a duplication (e.g., extensions of a previously found publication without further insights). This way, we excluded 69 publications (i.e., ⑥).

Classification prevention: After step 2), we decided whether the publication suggests a *recommendation* (i.e., subsequently abbreviated with the suffix `-R`), an *automated support* (i.e., subsequently abbreviated with the suffix `-AS`), or *both* as a prevention mechanism (i.e., using both suffixes) all of them targeting API misuse causes. In detail, we denoted a recommendation as support without software artifacts, such as suggesting an optimization of the software development process or a guideline. In contrast, we determined the prevention mechanism as automated support, when it represented a software artifact obtained via engineering research [RAB+20].

Classification validation: Subsequently, after steps 3) and 4), we decided whether the recommendation of automated support was *non-validated* (i.e., subsequently abbreviated with prefix `NV-`), *partially validated* (i.e., subsequently abbreviated with prefix `PV-`), or *validated* (i.e., subsequently abbreviated with prefix `V-`). We determined a recommendation or automated support as *non-validated* if no evaluation of the effectiveness of preventing API misuses was conducted. In case we denoted it partially validated, we found that only some parts of the recommendation or automated support were evaluated regarding the effectiveness of preventing API misuses using methods from empirical software engineering (e.g., developer studies, surveys, engineering research) [RAB+20]. In contrast, for a prevention mechanism denoted as *validated*, all mechanisms have been completely evaluated.

Mapping: In step ⑦, the main author mapped for each of the 411 publications the previously obtained classification to the *detailed root causes* (cf. Table 4.4). Note that in case multiple prevention mechanisms were reported, for instance, a `VAS` and an `NVR`, both kinds were counted. Moreover, single prevention mechanisms could map to multiple root causes. This way, we determined how many differently validated prevention mechanisms for each root cause exist. This procedure caused that the reported number represented the *number of different reported prevention mechanisms* and *not necessarily the number of different publications*.

Based on this procedure, we conducted a quantitative analysis of possible research gaps in API misuse prevention mechanisms. Moreover, we subsequently determined the most frequently discussed prevention mechanisms of each of the 11 general root causes. Note that a complete discussion of all 411 publications would be too fine-grained. Thus, for details, we refer to our replication package[6].

Results Meta-Analysis First, we analyzed the characteristics of the 411 publications to ensure a sufficient quality of the selection. Figure 4.14 depicts the absolute frequency of publications per year. We observed that the majority of publications stem from the years 2018 to 2023, while we found a significantly large set of publications from 2024 and also two publications from 2025. Note that the decrease in publications in 2024 and 2025 does not necessarily represent a decrease in the research interest in API prevention mechanisms but rather is an artifact of non-indexed publications within Google Scholar.

We also reviewed the venues of the publications and focused on those venues with at least ten publications within our final set in decreasing order in Table 4.7. We found that these represent publications from leading conferences[8] and journals[9] from software engineering. Note that these are conferences with special focuses on human interaction

[8] A and A* conferences according to the ICORE conference portal cf. https://portal.core.edu.au/conf-ranks/ last accessed: 2025/03/20

[9] listed in the top ten venues of Google Scholar category of *Software Systems* https://scholar.google.com/citations?view_op=top_venues&vq=eng_softwaresystems last accessed: 2025/03/20

Table 4.7.: Venues sorted by the number of included publications with at least ten publications in our reviewed set

Venue	#Publications
ACM/IEEE Int. Conf. Soft. Eng. (ICSE) and its *Softw. Eng. Pract. (ICSE-SEIP)* track	47+5
IEEE Trans. Softw. Eng. (TSE)	22
Springer Empir. Softw. Eng. (EMSE)	22
ACM Eur. Softw. Eng. Conf./Found. Softw. Eng. (ESEC/FSE)	19
IEEE Int. Conf. Softw. Maint./Evol. (ICSM/ICSME)	7+10
IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)	15
IEEE Int. Conf. Program Compr. (ICPC)	14
ACM Conf. Hum. Factors Comp. Syst. (CHI)	13
IEEE Int. Conf. Softw. Anal., Evol., Reeng. (SANER)	11
ACM Trans. Softw. Eng. Methodology (TOSEM)	11
Elsevier J. Syst. Softw. (JSS)	11
Elsevier J. Inf. Softw. Technol. (IST)	11

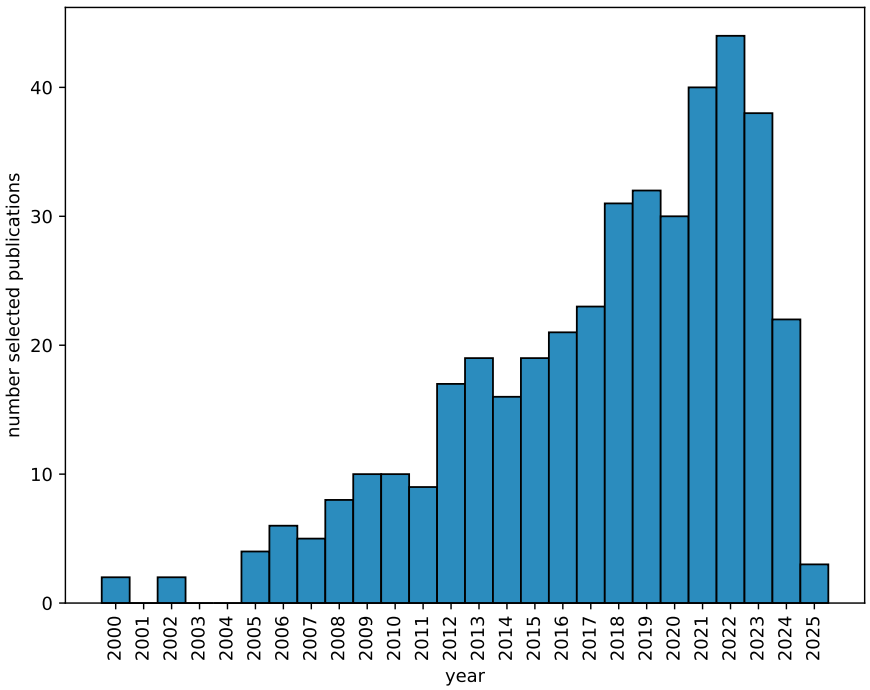


Figure 4.14.: Frequency of selected papers with API misuse prevention per publication year

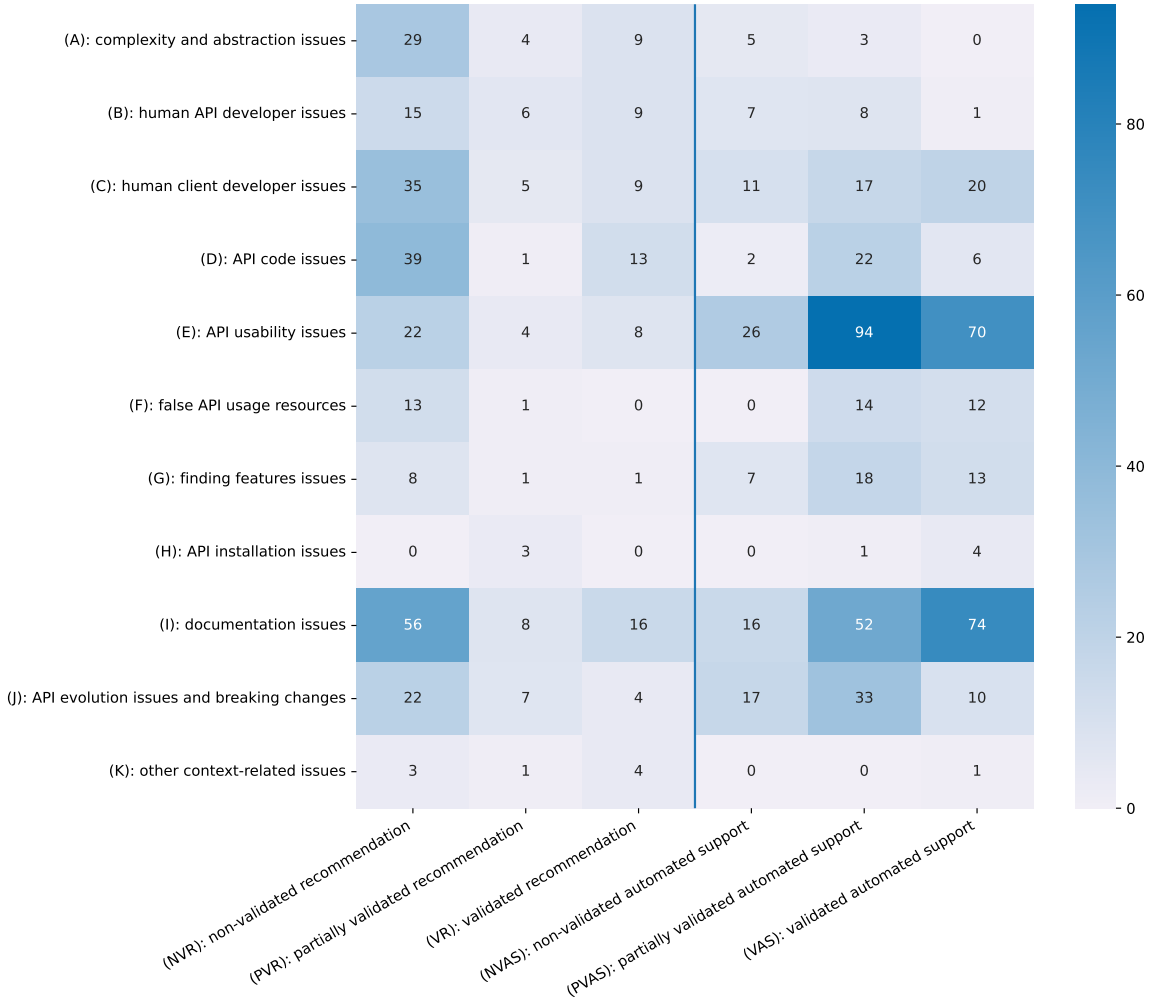


Figure 4.15.: Correlation of General Root Causes and Prevention Mechanisms

(i.e., CHI), program comprehension (i.e., ICPC), maintenance (i.e., ICSM/ICSME), and software analytics (i.e., SANER).

Due to its time frame encompassing recent research as well as due to high-ranked venues, we denote this set as *representative of software engineering research*.

Results Quantitative Mapping Prevention Mechanisms and Root Causes As discussed before, we mapped each single prevention mechanism (either recommendation or automated support) to one or multiple detailed root causes. A summary of this mapping of root cause groups with their prevention mechanisms is shown in Figure 4.15 as a heat map[10].

Based on our previous results, we know that (E) *API usability*, (I) *documentation*, (A) *complexity and abstraction*, and (D) *code issues* are the most reported root causes of API misuses (i.e., **Insight C&P-1**). This way, it was not surprising that, most frequently, `AS`-prevention was researched for (E) *API usability* (i.e., 94 `PVAS`s and 70 `VAS`s) and (I)

[10] the mapping to detailed root causes is shown in the appendix (cf. Figure A.2) as well as in the replication package in footnote 6 of this Chapter

API documentation issues (i.e., 52 PVAS and 74 VAS). (I) *API documentation* also had the largest number of VRs (i.e., 16) and PVRs (i.e., eight). In comparison, no VAS and only three PVAS were researched for (A) *complexity and abstraction*, while four PVRs and nine VRs were discussed for this issue. For (D) *API code issues*, we found the second most number of VRs (i.e., 13) and a single PVR as well as 22 PVAS and six VAS. In addition, we found more than ten PVAS or VAS for the root causes of issues with (C) *human client developer*, (F) *false API usage resources*, (G) *finding feature issues*, and (J) *API evolution and breaking changes*. Overall, we found the least number of prevention mechanisms for (H) *API installation issues* (i.e., overall eight) and (K) *other context-related issues* (i.e., overall nine).

Insight C&P-6 (RQ C&P-P): Most prevention mechanisms for API usability issues and API documentation issues

We found that in our investigated literature set, the root causes of (E) API usability issues (i.e., twelve partially or fully validated recommendations and 164 partially or fully validated automated support techniques) and (I) API documentation (i.e., 24 partially or fully validated recommendations and 166 partially or fully validated automated support) were most frequently targeted in API misuse prevention research.

Except for two causes (H) and (K), we noticed among all other causes, *more NVRs than VR and PVR* exist. This observation indicates that only a minority of prevention recommendations are actually evaluated, namely, in sum, 114 PVRs and VRs compared to 242 NVRs.

Insight C&P-7 (RQ C&P-P): Many non-validated recommendations

While we found many recommendations from API misuse prevention research, a majority of them (i.e., 242 of 336) were not validated using empirical methods of software engineering research.

In contrast, automated support was more frequently evaluated (i.e., in sum 211 VASs and 262 PVASs compared to 91 NVASs) among all with the exception of (A) complexity and abstraction, which had only three PVAS compared to five NVAS (e.g., tool ideas suggested by researchers).

Insight C&P-8 (RQ C&P-P): More validated and partially validated automated support

We found that a majority of automated support is partially or fully validated (i.e., 473 prevention mechanisms) in comparison to non-validated techniques (i.e., 91).

We also depicted the absolute frequency of PVR/PVAS and VR/VAS prevention mechanisms in Figure 4.16. Based on this, we observed that the root causes with issues regarding (C) human client developer, (D) API code, (E) API usability, (F) usage of API resources, (G) finding features, (H) API installation, (I) API documentation, and (J) API evolution and breaking changes were most frequently targeted with validated and partially validated automated support. For issues (A) complexity and abstraction, (B) human API developer, and (K) other context-related, research suggested more recommendations. Recall that due to the counting procedure (i.e., counting prevention mechanisms for detailed causes), a single prevention can be counted multiple times within a cause group.

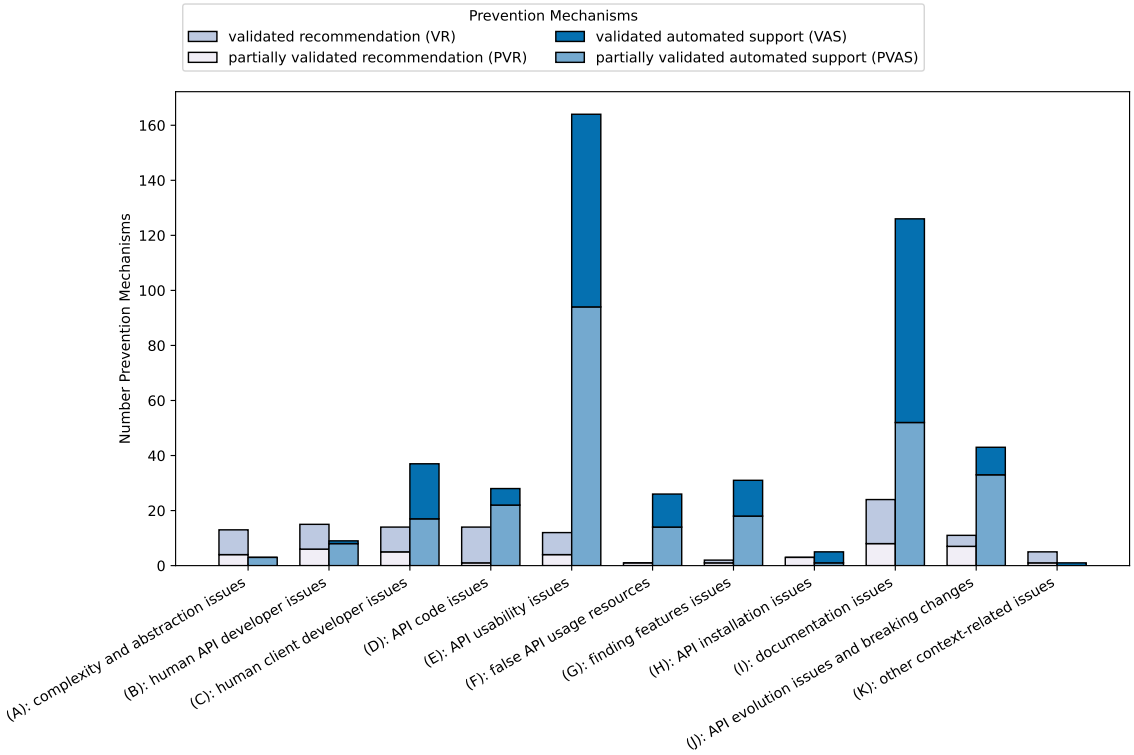


Figure 4.16.: Number of (partially) validated prevention mechanisms per root cause

Insight C&P-9 (RQ C&P-P): More automated support than recommendations in the set of validated and partially validated prevention mechanisms

We found that for all validated and partially validated prevention mechanisms overall (i.e., 473 automated support compared to 114 recommendations) as well as on the level of targeted root causes (i.e., eight out of eleven had more automated support than recommendations) research on API misuse prevention was focused on providing automated support.

Results Qualitative Discussion Prevention Mechanisms on Root Causes We present an overview of concrete prevention mechanisms of validated results (i.e., `VR` and `VAS`) according to the general root causes. For that purpose, we leveraged the mapping and re-consulted the summary texts referring to the respective validated prevention mechanism. We present a detailed summary of prevention mechanisms in our appendix (cf. Table A.3). Subsequently, we discuss the most important prevention mechanisms for each root cause.

(A) Complexity and abstraction issues: We found relatively few `VR`s for *complexity and abstraction issues*, and most of them stem from the domain of cryptographic APIs. Their focus was on design principles and guidelines [ZWH19], and the goal was to simplify the usage for common cases [FMGK24], provide simpler test modes [IKND16], or secure defaults [IKND16, FMGK24].

(B) Human API developer issues: *Human API developer issues* were less researched, particularly in comparison to client developers issues (i.e., root cause (C)). Most `VR`s focused

on improving the communication means between API and client developers, for instance, by establishing code review processes involving client developers [WBK21], applying certain API usability evaluation methods [FWZ10], and communicating up-to-dateness using badges in repositories [TZKV18]. A single VAS was given by a chatbot, supporting the API developer to adhere to functional requirements during the development of the API [SY21].

(C) Human client developer issues: Regarding *client API developer issues*, a typical VR, especially found for cryptographic APIs, was that priming client developers with task information (e.g., be aware of security issues) triggers the correct usage [NDT+17, DNRS21]. Another valid result was that pure copy and paste can lead to functional but not necessarily secure code [ABF+17]. More positive results were found using code examples but in combination with pair programming [SAKW21]. Thus, many VAS techniques focused on selection of relevant Q&A pages [FXK+19, WJZ+23, CPY+24] as well as automated in-code suggestions, which leveraged context information [NMH+24], related code examples [IHK17], usage templates [MWD24], and code generation [KRW+23]. Other VASs aimed to improve the usability of documentation by collecting scattered information [LLS+18] and shared common knowledge via annotations [HLH+22].

(D) API code issues: For *API code issues*, we found the VR to avoid linguistic antipatterns and non-universally accepted terms (e.g., from the application domain), particularly when naming API elements [ANBL18, MRARMB+18]. Other VRs focused on increasing the resilience of API code to fail due to internal errors [MRARMB+18], reporting the current build status [TZKV18], and applying Static Code Analysis (SCA) with more precise messages [TVBW21]. VASs were focused on the security domain in detail, providing security advice and warnings [NWA+17, GIW+18, LAH18].

(E) API usability issues: *API usability issues* were prominently targeted root cause. However, we only identified a few VRs, namely, not hiding API methods in helper classes [SM08] or preferring constructors for initialization over the factory pattern [ESM07]. More frequently, VASs were suggested, particularly on finding related or alternative API usages [HWM06, BDWK10, DR11, KAS13, IHK17, NKZ17, ZUR+18, LHT+22] or their automated generation [KAB20, SWZ+20, MWD24] together with a related explanation [YHWH24]. Another large field was the inference of idiomatic code examples or API usage patterns, which served as templates or sources of code suggestion techniques [MXBK05, DH09a, SHA14, BSZ+20, LPM+21, NDRDS+22]. Moreover, many techniques improved the usability of Q&A pages, in which API misuses were discussed [HAHH15, GZHK18, NGB21].

(F) False API usage resources: A large focus for the prevention of *false API usage resources* targeted the usage of Q&A pages. In detail, we only found VASs, which validated [FXK+19, HWL21] and summarized [WPWZ19, LHT+22, LCC+23, CPY+24] Q&A posts. Other techniques provided means to customize code examples from those discussion boards [ZYLK19, TS21]. An interesting research area was the automated detection of frauds. Frauds denote the up-voting of low-quality solutions, particularly when developers want to promote themselves [MUKS24].

(G) Finding features issues: For the root cause of *finding features issues*, namely, relevant API elements, we only found a single VR, namely, not hiding API methods in helper classes [SM08]. Instead, a larger set of VASs focused on improving code search techniques [EM21, KRW+23, BFH24] and providing recommender systems for APIs [SPK14, HXX+18] and libraries [CXL19, LZP+23]. Other techniques improved the findability of

API elements in the documentation by providing usage frequencies [SFYM09] as well as leveraging information from Q&A posts [UKR20] and API method similarities [NMVH23]. Moreover, Santos and Myers [SM17] contributed annotations for API code to enhance the API discoverability. When providing tooling for customizing and summarizing code examples from Q&A posts, an improved API exploration was observed [GZHK18, ZYLK19].

(H) Installation issues: We only found a few `VAS`s as prevention mechanisms for *installation issues*. Mostly, these were focused on specific domains, such as optimization techniques for Deep Learning (DL) APIs [WLY+24] or automated privacy configuration for mobile apps [SAI20]. Another technique resolved the issue of dependency incompatibilities [LWL+22].

(I) API documentation issues: As the second most targeted root cause, *API documentation issues* had many prevention mechanisms. Most frequently, we found, as `VR`, studies that validated the positive effect of code examples to explain API usage to client developers [WZF+12, SAKW21, BCM22, GMWI22] as well as leveraging so-called crowdsourced documentation, such as Q&A pages [DPCdAM16, TRJ+21]. Another `VR` suggested constructing documentation that supports different human learner types, namely opportunistic (i.e., with the goal of fast task solving) versus systematic learners (i.e., with the goal of obtaining a complete understanding of the API) [MSS20]. Ukrop et al. [UBv+22] validated the positive effect of highlighting potential misuses when applying the API. Since code examples had been considered as a positive characteristic of documentation, it was not surprising that there exist many techniques finding examples, templates, and patterns of API usage [MXBK05, HWM06, DH09b, BW12, KLHK13, IHK17, NKZ17, GZHK18, HGHH18, ZYLK19, BSZ+20, SXP+21, ZJR+21, WBI+23, BFH24]. Moreover, there exist techniques that leverage crowd-sourced information, such as Q&A webpages, to enhance the documentation [SFYM09, BDWK10, GFX+10, TR16, CYYZ19, WPWZ19, ZJR+21, WJZ+23, BFH24]. Another focus was the improvement of the search abilities and orientation in the documentation [RC15, LLS+18, WJZ+21, HLH+22, NMVH23]. During the creation of the documentation, some `VAS`s supported the process of writing the documentation [LWCK21, HMM23], while others strove for automated documentation generation [MM16b, UKR20, UKR21, GMBG+23]. More specific techniques aimed at the creation of tutorials within documentation [KRW+23, NMH+24] and provided interactive mechanisms in the documentation [DH09a, NR25]. Finally, to ensure the quality of the documentation, `VAS`s detected inconsistencies between source code and the documentation[MLK19, ZWY+20] and provided automated assessment [BCH+23].

(J) API evolution issues and breaking changes: Interestingly, while being a prominently found issue, `VR`s for API evolution in our investigated literature set were rare. We found a single work that validated the effect of semantic versioning, namely indicating by the version number the degree of change and possible breaking changes [RvV17]. Most `VAS`s suggested the application of automated API migration techniques for code [SPK14, GAQ+18, LSC22, DND+25] as well as its documentation [LWCK21, HMM23]. Other techniques aimed to detect incompatibilities when libraries were upgraded [NDBB20, ZZW+23]. Two other `VAS`s suggested alternative libraries [CXL19] or automatically generated compatibility code for the library [DNMJ08].

(K) Other context-related issues: For the last root cause (K) *other context-related issues*, we only found a few `VR`s and `VAS`s. In detail, most frequently, we found the `VR`

of using statically over dynamically typed programming language [EHRS14, PHR14], but this effect was diminished in case documentation was not available [MHR+12]. Other recommendations focused on software development characteristics for the specific domain of smart contracts [WXL+21]. As `VAS`, we identified a technique to fix license issues in forked projects [HXC+24].

Insight C&P-10 (RQ C&P-P): Different degrees of prevention mechanisms for API misuse root causes

We found that (1) for each considered root cause, we found prevention mechanisms, (2) however, the number and degree of prevention mechanisms varied strongly, (3) and some techniques (e.g., automated support finding related code examples for improving API usability and extending API documentation) were more prominently researched, and (4) some prevention mechanisms targeted multiple root causes (e.g., finding related code examples and providing summaries of Q&A pages).

Influence of security APIs: Since we noticed many prevention mechanisms related to the domain of security and cryptographic APIs, we checked how many papers are validated in the security or cryptographic domain. This is important, since certain prevention mechanisms could be more suitable in this specific domain and, thus, could bias the generality among software engineering research. For this purpose, we search in all 411 publications whether their titles contain the prefix `secur-` or `crypto-`. We found 53 publications. Moreover, we checked the publishing venues and found 41 publications from 19 different security-related venues. Thus, we obtained a proportion of $10 - 12.9\%$ of publications on prevention mechanisms stemming from the security domain. For all publications with validated prevention mechanisms (i.e., `VR` and `VAS`), which represent 131 publications in our set, we found 19 of them containing the previously named prefixes in their titles. Fifteen of these publications stemmed from 10 different security venues. These represented a proportion between $11.4 - 14.5\%$ of all publications with validated prevention mechanisms. From past research, we observed a rough notion on the proportion of security research in software engineering, which, depending on the actual measurement (i.e., proportion of publications vs. proportion of topics within publications), ranges between 1%-2% [GVR02, GDCS22]. Thus, we identified the influence of the security domain as small but not negligible in comparison to software engineering in general.

Insight C&P-11 (RQ C&P-P): Small but not negligible set of recommendations and automated support from the security domain

We observed a set between $10 - 14.5\%$ of publications with concrete prevention mechanisms stemming from the domain of security and cryptographic APIs and were validated in this specific domain.

Implications In this section, we surveyed the potential prevention mechanisms targeting the root causes identified in Section 4.2 to answer **RQ C&P-P**. We found that our assumed *recommendations* and *automated support* represent a sufficient classification to represent state-of-the-art prevention mechanisms. Moreover, we analyzed to which degree these prevention mechanisms were validated and which root causes of API misuses they target. Based on these results, researchers obtain an overview of which root causes are

most frequently analyzed with which prevention mechanisms (i.e., **Insight C&P-6** and **Insight C&P-10**) and which prevention mechanisms require further empirical evaluation (i.e., **Insight C&P-7**, **Insight C&P-8**, and **Insight C&P-9**). Practitioners could apply our overview (i.e., in Table A.3) to find a set of validated prevention mechanisms (i.e., **Insight C&P-6** and **Insight C&P-10**). Moreover, researchers and practitioners from the security domain should consider specific prevention mechanisms evaluated in the security domain (**Insight C&P-11**).

4.3.3. Recommendations for Research of API Misuse Prevention

Methodology In this section, we map the results of API misuse root causes (cf. Section 4.2.2) and API misuse prevention mechanisms (cf. Section 4.3.2) to obtain indicators for future research on root causes and their prevention mechanisms. For this purpose, we obtain a mapping of the number of publications discussing root causes with the number of prevention mechanisms.

In detail, first, we counted the number of publications for each detailed root cause as well as for each general cause (cf. Table 4.4). Then, we normalized each count by dividing it by the largest number (i.e., the most frequently discussed root cause). This way, we obtained a measurement of the *relative research effort* denoted as rr_{effort} for each root cause. This value represents the relative effort on one root cause compared to all others, namely, $rr_{effort} = 0$ denotes *no research*, while $rr_{effort} = 1$ denotes the largest relative research effort. Similarly, we obtained the rr_{effort} on the prevention mechanisms for each detailed root cause and general cause. However, we only considered those prevention mechanisms, which we denoted in Section 4.3.2 as partially validated or validated (i.e., `PVR`, `VR`, `PVAS`, or `VAS`). Note that by construction, since there will always be a root cause with the largest number of publications or suggested prevention mechanisms, there will always be one entry with $rr_{effort} = 1$.

We emphasize that rr_{effort} only represents the research effort in our analyzed literature and not necessarily represents the *true* value for *entire research domain of API misuse causes and prevention mechanisms*. Nevertheless, based on the descriptive data of the literature venues (cf. Sections 4.2.2 and 4.3.2) and the number of relevant literature (i.e., 65 publications for root causes and 296 from 411 publications for prevention mechanisms), we identify these sets as representative.

Finally, we discuss the differences to denote potential research directions (e.g., those root causes frequently reported but with few prevention mechanisms).

Results We present the results of the rr_{effort} for publications on root causes and suggested prevention mechanisms for each general cause in Figure 4.17. Note that this figure encompasses both prevention mechanisms, namely *recommendations* and *automated support* (i.e., both either validated or partially validated). In Figure 4.18, we separately considered the research effort on recommendation (cf. Figure 4.18a) and validated automated support (cf. Figure 4.18b). Moreover, we present in our appendix (cf. Section A.1.6) a fine-grained comparison of the relative research effort based on the detailed root causes (cf. Figure A.3, Figure A.4, and Figure A.5). Note that the rr_{effort} of the root causes is the same for each depiction (i.e., grey bars in Figure 4.17 and Figure 4.18) and we only varied the prevention mechanism (i.e., blue bars).

In general, we observed that the most frequently discussed root causes *(E) API usability issues* (i.e., $rr_{effort} = 1$) and *(I) documentation issues* (i.e., $rr_{effort} \approx 0.81$) also obtained

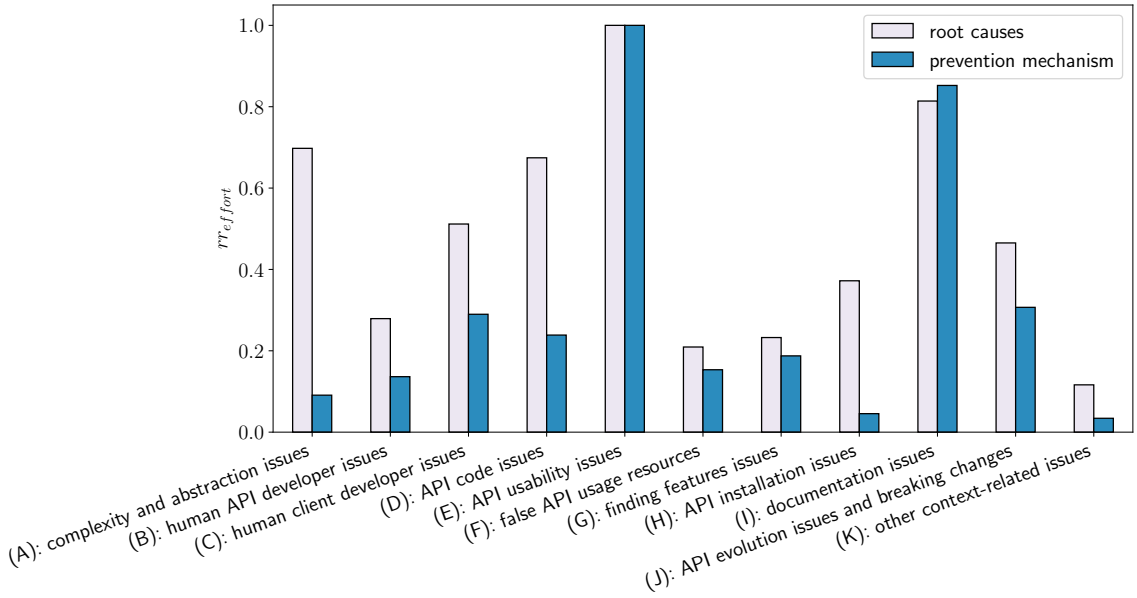


Figure 4.17.: Comparison of relative research between relative research effort (rr_{effort}) on root causes and prevention mechanisms in general

the largest relative research effort regarding the prevention mechanisms with an equal or larger rr_{effort}, namely, $rr_{effort} = 1$ and $rr_{effort} \approx 0.85$, respectively. For all other general causes, the rr_{effort} is smaller for prevention mechanisms. Most prominently visible is the difference for the root causes *(A) complexity and abstraction issues* having a more than seven times smaller rr_{effort} on prevention mechanisms compared to its root causes (i.e., $rr_{effort} \approx 0.09$ vs. $rr_{effort} \approx 0.7$). We also observed this behavior for *(H) API installation issues* with a more than eight times smaller rr_{effort} (i.e., $rr_{effort} \approx 0.05$ vs. $rr_{effort} \approx 0.37$). Moreover, we found that the frequently discussed root causes *(D) API code issues* (i.e., $rr_{effort} \approx 0.67$), *(C) human client developer issues* (i.e., $rr_{effort} \approx 0.51$), and *(J) API evolution issues and breaking changes* (i.e., $rr_{effort} \approx 0.47$) also had a smaller rr_{effort} on prevention mechanisms.

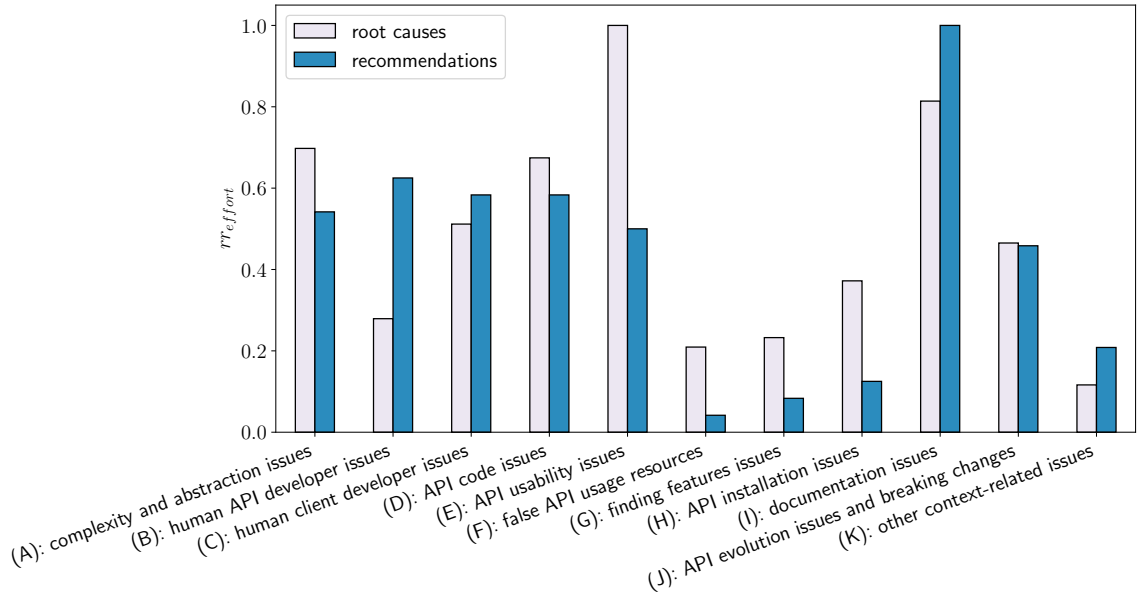
Insight C&P-12 (RQ C&P-S): Research potential on prevention mechanisms for a subset of root causes

We found, based on the conducted research on root causes, future research should concentrate the research effort on prevention mechanisms targeting (A) complexity and abstraction issues, (B) human API developer issues, (C) human client developer issues, (D) API code issues, (H) API installation issues, and (J) API evolution issues and breaking changes.

When considering the groups of validated/partially validated recommendations and automated support separately (cf. Figure 4.18), we observed that the rr_{effort} on recommendations is different from those of automated support.

The rr_{effort} on recommendations was larger in comparison to the overall prevention mechanisms for all but *(E) API usability issues*, *(F) false API usage resources*, and *(G) finding features issues*. For three causes (i.e., *(B) human API developer issues*, *(C) human*

(a) Only recommendations



(b) Only automated support

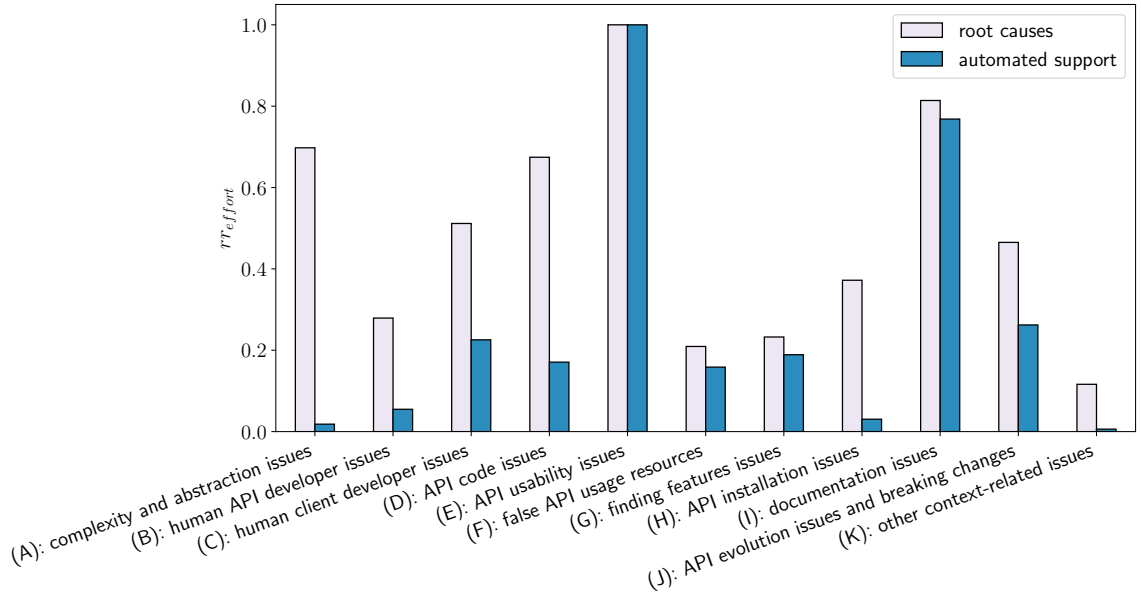


Figure 4.18.: Comparison of relative research between relative research effort (rr_{effort}) on root causes with recommendations and automated support

client developer issues, and *(I) documentation issues*), the rr_{effort} on recommendations expanded that of the root causes. Interestingly, the rr_{effort} on recommendations for *(E) API usability issues* was only half of the rr_{effort} compared to automated support. Nevertheless, we point out that these only represent relative numbers, as observed in the previous Section. Particularly, from **Insight C&P-9**, we know that the absolute number of partially validated and validated automated support is much larger than those of recommendations. Since we also observed a large set of non-validated recommendations (i.e., **Insight C&P-7**), future research should concentrate on validating these recommendations, particularly targeting the root causes of *(E) API usability issues*, *(F) false API usage resources*, *(G) finding feature issues*, and *(H) API installation issues*. In detail, future research could validate those NVRs from Figure 4.15 (or Figure A.2 in the appendix) or suggest and validate own recommendations for causes, for which no NVRs exist in our results (e.g., *(H) API installation issues*).

Insight C&P-13 (RQ C&P-S): In general, more research effort on recommendations with some particular focuses

We found that (1) the overall number of partially and fully validated recommendations was low. Thus, research should concentrate on validating the set of given recommendations. (2) A special focus should be put into recommendations regarding the root causes of (E) API usability issues*,* (F) false API usage resources*,* (G) finding feature issues*, and* (H) API installation issues*.*

In contrast, the rr_{effort} on automated support (cf. Figure 4.18b) was similar compared to the results of the prevention mechanisms. However, we observed for all root causes but two (i.e., *(E) API usability issues* and (F) false API usage resources) a lower rr_{effort} on automated support compared to the combined view on mechanisms. Moreover, the research effort on *(I) documentation issues* was comparable to the effort on the analysis of its cause. Thus, future research should aim for more automated support to target *(A) complexity and abstraction issues* and *(D) API code issues*. For both, techniques could target a better design or navigation through complex APIs or provide SCA techniques to find issues in the library code. Regarding *(B) human API* and *(C) client developer issues*, the potential benefit from automated support is debatable. Some root causes cannot be easily fixed with automated techniques, such as missing education on API design (i.e., the detailed root cause (B-3)) or targeting mindset issues of client developers (i.e., the detailed root cause (C-4)). In other cases, such as communication issues between API and client developers (i.e., detailed root cause (B-1)), automated communication means can be valuable features.

Insight C&P-14 (RQ C&P-S): Research effort on automated support only with a special focus

We found that (1) research effort on automated support should be put into (A) complexity and abstraction issues *and* (D) API code issues*, which have more potential to benefit from automated techniques, while (2)* (B) human API *and* (C) client developer issues *may only benefit from automated support techniques for a few detailed root causes.*

Insight C&P-15 (RQ C&P-S): Most current research effort was put into preventing API usability and documentation issues

Overall, we found that the root causes of (E) API usability issues and (I) documentation issues were most prominently researched regarding their causes as well as their prevention mechanisms.

Finally, research should further regard empirical or observational studies on less frequently researched root causes. A major need is the root cause of *(B) human API developer issues*, which has almost half of the research effort on its cause (i.e., $rr_{effort} \approx 0.28$) compared to *(C) human client developer issues* (i.e., $rr_{effort} \approx 0.51$). This way, a better comprehension of the needs and problems during API development can be achieved. Moreover, we also motivate research efforts on both causes and prevention mechanisms on *(F) false API usage resources*, *(G) finding features issues*, and *(H) API installation issues*, since all have a small rr_{effort} (i.e., below 0.37).

Insight C&P-16 (RQ C&P-S): Next to prevention mechanisms, research should further investigate root causes with previously a low research focus.

We found that (1) researchers should further analyze the root cause of (B) human API developer issues and (2) (F) false API usage resources, (G) finding features issues, and (H) API installation issues as they require more research on both root causes and prevention mechanisms.

Implications In this section, we analyzed the research effort on root causes of API misuses in comparison with their prevention mechanisms to answer **RQ C&P-S**. This way, we could assess which root causes, as well as which prevention mechanisms targeting these root causes, require further research. Thus, we suggest future research directions on prevention mechanisms (i.e., **Insight C&P-12**), particularly on recommendations (i.e., **Insight C&P-13**) and automated support (i.e., **Insight C&P-14**) as well as suggestions for studies on their root causes (i.e., **Insight C&P-16**). Nevertheless, we also found that some root causes are well-researched (i.e., **Insight C&P-15**), which provides insights for practitioners on the applicability of these results as well.

4.3.4. Threats to Validity

Regarding the research questions **RQ C&P-P** and **RQ C&P-S**, which we evaluated in this section, we discuss potential threats to validity.

Internal Validity Even though we applied methods from SLR, such as snowballing, our selected publications could suffered from selection bias. This bias could be caused by the main author's subjective view when selecting publications based on their titles or due to the time at which the snowballing was conducted (i.e., different search results when applying Google Scholar [KLvN+20]).

Another issue was that the manual mapping of prevention mechanisms to the root causes, as well as the classification of the prevention mechanisms, was solely done by the main author. Additionally, the naming of the root causes, as well as their descriptions, could also influenced whether there existed a mapping or not. This way, even though conducted

systematically, this mapping and classification could be biased by the main author or could contain errors. For this purpose, we provide all data together with their analysis scripts in a replication package[6].

Our measurement of relative research effort (rr_{effort}) depicts only the number of publications and suggestions relative to the other publications and suggestions and does not express the effectiveness of the discussed prevention mechanisms or the severity of the root causes. This way, it depicts all prevention mechanisms as equally effective and the root causes as equally severe, which could be false in reality.

External Validity Since our results were based on the root causes identified before, we might also miss prevention mechanisms for potentially unknown or unreported root causes.

Due to the influence of security-related publications (cf. **Insight C&P-11**), results on the prevention mechanisms could be partially limited to this domain. While we obtained relatively new publications, we emphasize that the results are limited to the current time frame and report the state of the literature for this time frame.

Moreover, the expressiveness of rr_{effort} is limited to the selected set of publications and the presented mapping. Thus, a different mapping and potential other publications (e.g., found due to another not considered root cause) can deviate the obtained result on the research effort.

4.4. Summary API Misuse Root Causes & Prevention

Summary In this chapter, we targeted the research question of whether current *prevention mechanisms target the root causes of API misuse* (i.e., **RQ C&P**). We answered this general question by conducting two subsequent SLR-like processes [KC07, Woh14].

First, we searched empirical studies evaluating the root causes of API misuse to answer **RQ C&P-C** on *common root causes*. We found a set of 65 publications, from which we extracted their analyzed root causes and their applied research methodologies using open coding from qualitative research [Fli14].

Second, we used this set to retrieve 411 publications, which discussed potential prevention mechanisms to target **RQ C&P-P** on *the state-of-the-art prevention mechanisms and processes*, subsequently referred to as *prevention mechanisms*. We summarized the prevention mechanisms and classified each based on their type and validation status. Then, we mapped each prevention mechanism to its targeted root cause, which we determined in the previous review.

Finally, we analyzed potential research gaps by this mapping of API misuse causes and their prevention mechanisms to determine *whether the state-of-the-art prevention is sufficient* (cf. **RQ C&P-S**).

Contribution RQ C&P-C We provided an overview of a diverse set of root causes of API misuses structured into eleven categories with further 44 detailed root causes (cf. **Insight C&P-1** 'Diverse Root Causes' on page 59). This overview denotes an extension to state-of-the-art reviews (cf. Section 4.2.1) while encompassing all previously discussed causes. We also determined that these root causes have interdependencies, a phenomenon – to the best of our knowledge – not analyzed in the scientific literature. Therefore, we suggested the concept of *views* (cf. **Insight C&P-2** 'Root Cause Relations by Views' on page 63) to formulate hypotheses, which serve as a basis for further research on API misuse causes.

A further result is that we mapped the studies to their research methodologies. This way, we obtained an overview of the typical structure of empirical studies on API root causes (cf. **Insight C&P-3** 'Typical Empirical Study Structure 'on page 63). Moreover, this enabled the assessment of the scientific evidence of single root causes. For instance, we found that a majority of root causes were evaluated using qualitative methodologies (cf. **Insight C&P-4** 'More Qualitative Studies' on page 67) as well as that only single root causes were validated based on multiple, different methodologies (cf. **Insight C&P-5** 'More Diverse Studies for API Evolution Issues' on page 67).

Thus, we conclude for **RQ C&P-C**:

RQ C&P-C What are the common root causes of API misuses?

We identified (1) a variety of root causes grouped into eleven main root causes with 44 sub-root causes and (2) potential interdependencies among them, which we hypothesized in the form of views. We further determined (3) the typical structure of empirical studies on root causes and (4) found that many studies on root causes are limited to qualitative studies.

Contribution RQ C&P-P Regarding the prevention mechanisms, we found *recommendations* and *automated support*, however, with a different focus on the previously discussed root causes (cf. **Insight C&P-6** 'Most prevention mechanisms for API usability issues and

API documentation issues' on page 76). Based on the analysis of the validation status, we found that the majority of recommendations have not been validated (cf. **Insight C&P-7** 'Many non-validated recommendations' on page 76). That means *even though research on API misuse prevention has a large variety of ideas to avoid the root causes, it lacks sufficient empirical studies to validate their effectiveness.* In contrast, much more research effort and evidence is provided by automated support based on software artifacts (cf. **Insight C&P-8** 'More validated and partially validated automated support' on page 76 and **Insight C&P-9** 'More automated support than recommendations in the set of validated and partially validated prevention mechanisms' on page 77).

We also provided a qualitative overview of prevention mechanisms evaluated in the scientific literature. This overview supports researchers and practitioners alike (cf. **Insight C&P-10** 'Different degrees of prevention mechanisms for API misuse root causes' on page 80). Additionally, we found that specifically, the domain on security APIs boosted the research on API misuse prevention (cf. **Insight C&P-11** 'Small but not negligible set of recommendations and automated support from the security domain' on page 80).

Therefore, we answer **RQ C&P-P** as follows:

RQ C&P-P What are the state-of-the-art prevention mechanisms targeting API misuses?

We found that (1) there exist different prevention mechanisms which can be abstracted as recommendations and automated support, however, (2) research is partially focused on individual root causes and (3) typically concentrated on validated automated support instead of (4) recommendations, for which a majority is not validated.

Contribution RQ C&P-S Based on the previous results and our mapping, we provided a mean to assess the current state of the research and suggested further research directions, namely, which root causes require prevention mechanisms (cf. **Insight C&P-12** 'Research potential on prevention mechanisms for a subset of root causes' on page 82). In more detail, we suggested research directions for validating recommendations (cf. **Insight C&P-13** 'In general, more research effort on recommendations with some particular focuses' on page 84) and automated support (cf. **Insight C&P-14** 'Research effort on automated support only with a special focus' on page 84). Moreover, we also analyzed which root causes were targeted more often by prevention mechanisms. This way, practitioners can identify potentially applicable mechanisms (cf. **Insight C&P-15** 'Most current research effort was put into preventing API usability and documentation issues' on page 85). Finally, we also suggested the need for further empirical studies on the root causes themselves to identify effective means to prevent them (cf. **Insight C&P-16** 'Next to prevention mechanisms, research should further investigate root causes with previously a low research focus.' on page 85).

This way, we provide the following answer to **RQ C&P-S**:

RQ C&P-S Does state-of-the-art research on prevention mechanisms sufficiently target root causes of API misuses?

In its current form, state-of-art research prevention only partially targets the API misuse root causes due to a lack of (1) validated recommendations, (2) targeted root causes, and (3) scientific evidence for a subset of root causes.

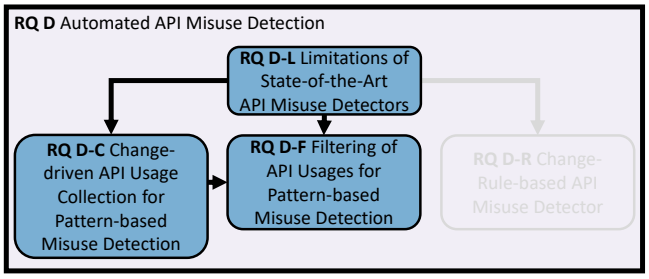
5

Improving Pattern-Based API Misuse Detection

This chapter is based on publications from the author together with other colleagues published and presented at the International Workshop on Software Mining (SoftwareMining) 2018 [NHO18], published in the Automated Software Engineering Journal in 2021 [NHSO21][1], and a non-peer-reviewed pre-print [NBKO22] discussed in the poster session of the Summer School on Security Testing and Verification 2022

In this chapter, we present our advances in improving pattern-based Application Programming Interface (API) misuse detectors through new supporting techniques. Misuse detection is necessary when the root causes (cf. Chapter 4) cannot be prevented. In this chapter, we discuss the improvement of existing pattern-based misuse detectors, while in Chapter 6, we discuss a new technique for API misuse detection. Patterns used for misuse detection will be subsequently leveraged in the automated repair of misuses (cf. Chapter 7).

5.1. Methodology and Structure



We target the challenge of improving pattern-based misuse detection within this chapter and thus answering research question **RQ D**. This encompasses the sub-research question on limitations of state-of-the-art misuse detectors (i.e., **RQ D-L**). We leverage change information to incorporate API misuse detection into realistic use cases (i.e., **RQ D-C**) and apply search and filter strategies to improve donor code quality for API specification mining, and thus to increase precision of the detection (i.e., **RQ D-F**).

We address **RQ D-L** in Section 5.2 using a Systematic Literature Review (SLR) methodology [KC07, RAB+20] to assess the state-of-the-art of API misuse detectors with a focus on their current limitations for practical application. Based on this overview, we also select a pattern-based misuse detector, which serves as a comparison to validate the subsequent improvements in patterns-based detection.

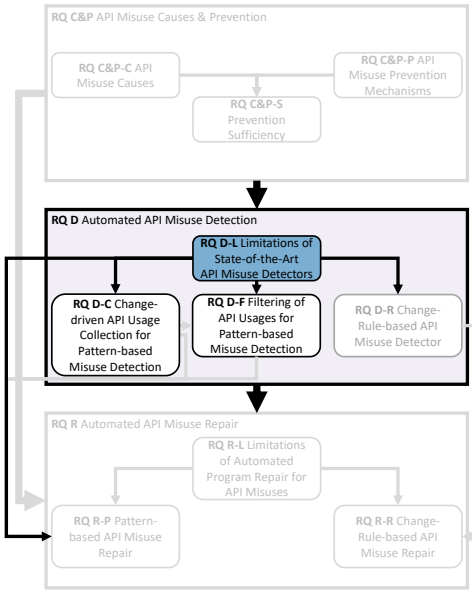
[1] This publication is partially based on the bachelor thesis "Extraktion relevanter API-spezifischer Informationen zur automatischen Korrektur von Softwarefehlern" by Kevin Michael Schott in 2018

Subsequently, we target the questions **RQ D-C** and **RQ D-F** by applying engineering research [RAB+20]. In detail, for **RQ D-C**, we present a concept (cf. Section 5.3) and a software artifact implementing the change information extraction. Then, we discuss our evaluation of this artifact with benchmarks of API misuses in an experimental setting (cf. Section 5.4), and show the applicability of change information for practical scenarios (cf. Section 5.5). Using the same concept, software artifact, and datasets (cf. Sections 5.3, 5.4, 5.5), we present the results of the qualitative and quantitative evaluation of the detection ability and the comparison of the impact of search and filter strategies to a state-of-the-art pattern-based misuse detector (cf. Section 5.5).

For all experimental results, we discuss potential threats to validity as suggested by Siegmund et al. [SSA15]. We summarize all results and their impact in Section 5.6.

5.2. Limitations of State-of-the-Art API Misuse Detection

In this section, we present general terms on API misuse detection, summarize the state-of-the-art of misuse detectors, emphasize their current limitations, and justify our selection of misuse detectors for experimental comparison.



5.2.1. General Terms on API Misuse Detection

We denote API misuse detectors as techniques to *recognize* and *locate* API misuses. In particular, we are interested in automated techniques since these enable software developers to efficiently find misuses in their code. From an abstract perspective, an API misuse detector requires a description of the correct behavior of an API usage, namely, a *specification*. Automated detection techniques apply *inference* technologies to obtain these specifications, for instance, from other correct API usages. Then, the detection validates an actual API usage against this specification. If the validation fails, we denote this usage as a *violation of the specification*, and the detection reports the API usage as a potential misuse, namely, a *positive* result of the detection. In case this detected misuse is indeed a real misuse, we refer to it as *true positive (tp)*. If, however, this usage is actually correct, we denote the detection as *false positive (fp)*. Similarly, we denote a real misuse, which is not detected as *false negative (fn)*, and a correct usage, which is not detected as *true negative (tn)*. To compare the effectiveness of API misuse detectors, we usually compare the *precision* and *recall*, which is based on the number (i.e., #) of tp, fp, and fn according to Equation 5.1 and Equation 5.2. *Precision* measures how many of the detected misuses are indeed real misuses and thus indicates the *trustworthiness* of positive results. *Recall* determines how many misuses in comparison to all present misuses are detected and thus describes the *exhaustiveness* of the detection.

$$precision = \frac{\#tp}{\#tp + \#fp} \tag{5.1}$$

$$recall = \frac{\#tp}{\#tp + \#fn} \tag{5.2}$$

We evaluate misuse detectors by measuring precision and recall, and by validating against *benchmarks* as *datasets* of known correct API usages and known misuses, which we refer to as *ground truth*.

5.2.2. State-of-the-Art API Misuse Detectors

Collecting the State-of-the-Art There exists much research effort in the domain of API misuse detection. To obtain an overview, first, we collected surveys summarizing the state-of-the-art on API misuse detectors. For this purpose, we reviewed the comparative studies by Amann et al. [ANN+19a, ANN+19b] in which they evaluated *five misuse detectors*, namely DMMC [MBM10], GrouMiner [NNP+09b], Jadet [WZL07], and Tikanga [WZ11] as well as their own misuse detector MUDetect [ANN+19b]. Moreover, we consulted the survey by Robillard et al. [RBK+13] on API pattern inference and selected potential misuse detectors (i.e., we selected publications from Table 1, p. 615, Table 3, p. 619, and Table 4, p. 627 in the work by Robillard et al., which are assigned to the goal 'Bug Detection' or 'Bug Finding'). We filtered these 24 publications from the work by Robillard et al. to avoid a possibly huge number of misuse detectors when using these publications for further forward snowballing. In particular, we only selected publications which

- are peer-reviewed;
- present an API misuse detection technique;
- evaluate (or which are evaluated by another publication) the precision and/or the recall of API misuse detection with more than one real-world project (i.e., excluding case studies).

Particularly, the last criterion ensured stronger scientific evidence due to a larger evaluation while still obtaining a sufficiently large number of detectors to derive a good picture of the state-of-the-art. Through this procedure, we found *nine additional misuse detectors* discussed by Robillard et al. [RBK+13] compared to the work by Amann et al. [ANN+19a], as well as a study evaluating different misuse detectors.

Based on this initial set of 14 publications, we applied the scholarly search machine Google Scholar[2] to conduct a forward snowballing [KC07] (i.e., by using the 'cited-by' functionality) to retrieve papers published in 2020 or later that cited these publications. Moreover, we found another misuse detector independently from this snowballing procedure, which targeted another programming language (i.e., EMDetect [ÇM18]) not used so far and thus worth being included in this set. We selected papers indicating by their titles that they contributed an API misuse detector and that satisfied the previously mentioned criteria. We found 16 additional misuse detectors, summing up to *30 different API misuse detectors*. Moreover, we derived *41 single evaluations reporting misuse detection results*. We did this iteratively until no further publications were found. Note that due to the technical limitations of scholarly search machines [KLvN+20], the search results may differ when being replicated. However, our goal was to obtain an *overview of relevant work* on API misuse detection rather than an *exhaustive search result*.

[2] https://scholar.google.com/ last accessed: 2024/02/16 and due to a missed publication (i.e., Doc2Spec [ZZXM09]) in the initial set, we re-conducted the forward snowballing for this paper on 2025/01/29

Table 5.1.: Meta Information and Characteristics of Misuse Detectors

Short Term	Description	Potential Values
L|F	Programming Language/Framework	name of programming langauge and/or framework name which is targeted by the miuse detector
A	Available Replication Package	indicator whether there is a replication package (✓), whether the replication package is not available anymore (✗*), whether there is no replication package (✗)
S	Source Data for Specification Inference	client code (C_C), API library code (C_A), changes in client code (C_Δ), API documentation (D); other external sources (E)
C	Collection Technique of Client code	not applicable ($-$), not specified (?), code from intra-project setting (I), code from cross-project setting ($\times$) with API-specific information (e.g., class, method, or parameter type names) ($\times_A$) or with other information ($\times_O$)
I	Inference Technique(s)	Static Code Analysis (SCA), Dynamic Code Analysis (DCA), Frequent Pattern Mining (FPM), Active Learning (AL), Deep Learning (DL), Hidden Markov Model (HMM), Natural Language Processing (NLP)
P	Post-processing	conducts filtering or ranking on the reported violations (✓), no dedicated post-processing (✗)

Classification of API Misuse Detectors For each relevant publication, we collected meta information on the detector and information to classify the technique (cf. Table 5.1) itself as well as the reported results (i.e., used datasets and results on precision and recall cf. Table 5.5).

We present the retrieved API misuse detectors by splitting them into separate classes based on their specification type. We distinguish between:

- *Single usage, explicit specification (SES)*: The detector uses a specification representing a single correct API usage, and its representation explicitly describes API code elements (e.g., association rules of method call pairs). For instance, an SES denotes that for the API `java.util.Iterator` in Java, a call of `hasNext()` is required before calling `next()`.

- *Multiple usage, explicit specification (MES)*: The detector uses a specification representing multiple correct API usages in one representation and explicitly describes API code elements (e.g., finite state automata describing multiple possible execution paths). For instance, an MES denotes that for the API `java.util.Iterator` in Java, a call of `hasNext()` is required before calling `next()`, but it also allows the iterated object to be checked via the `size()`-method.

- *Implicit specification (IS)*: The detector uses a specification representing correct API usages, and the code elements are implicitly handled in its model (e.g., a statistical

Table 5.2.: Overview of API misuse detectors with single usage, explicit specifications (SES). Labels according to Table 5.1

Misuse Detector	L|F	A	S	C	I	P
APDetect [WXQ23]	Java	✗	C_C	?	SCA, FPM	✓
APISan [YMS+16]	C, C++	✓	C_C	?	SCA, FPM	✓
APP-Miner [JWL+24]	C	✓	C_C	×	SCA, FPM	✓
CAR-Miner [TX09b]	Java	✗*	C_C	$\times_A$	SCA, FPM	✓
CL-Detector [ZCSZ21]	Java	✓	C_C, C_A	$\times_A$	SCA, FPM	✓
CPAM [LCP+21]	Java	✓	C_C, C_Δ	$\times_O$	SCA, FPM	✓
DMMC [MBM10]	Java	✗*	C_C	I	SCA, FPM	✓
FuzzyCatch [NVN20]	Java, Android	✓	C_C	$\times_O$	SCA, FPM	✓
GrouMiner [NNP+09b]	Java	✗	C_C	I	SCA, FPM	✓
Jadet [WZL07]	Java	✓	C_C	I	SCA, FPM	✓
MUDetect [ANN+19b]	Java	✓	C_C	I,$\times_A$	SCA, FPM	✓
PR-Miner [LZ05]	C	✗	C_C	I	SCA, FPM	✓
SpecCheck [NK11]	Java	✗*	C_C	?	DCA,FPM	✓
Tikanga [WZ11]	Java	✗*	C_C	I	SCA, FPM	✓

model measuring similarity to previously seen correct API usages). For instance, given the API `java.util.Iterator` in Java and its method `next()`, an IS computes a large probability that `next()` is preceded by the method `hasNext()`.

While this classification had been developed ad-hoc when searching and analyzing the related work, we found it helpful to understand certain design decisions of misuse detectors. Note that we do not claim this classification to be complete.

Specifications of SES detectors usually do not generalize well since they are naturally restricted to a single use case. Therefore, typically, these detectors apply *post-processing* strategies on the detected misuses, such as filtering and ranking techniques using measures that respect, for instance, alternative, applicable specifications. This way, possible false positives are discarded from the reported results.

MES detectors, on the other hand, merge multiple, possibly alternative, correct API usages within one representation. This way, these detectors avoid heavy post-processing, however, increasing the complexity of the representation of the specification and, thus, the complexity of the inference technique.

Note that we denote SES and MES as *explicit* since both contain direct information on API code elements. This way, we expect that the specification itself can help developers to derive correct API usages from it due to directly mapping to known API code elements.

In contrast, IS detectors apply *implicit* specifications, which we conjecture as hardly directly applicable for developers without having a mechanism or additional information to derive API code. Moreover, one cannot directly judge whether the specification is based on a single usage or multiple alternative usages. Implicit specifications usually require large amounts of input data (e.g., when applying machine learning algorithms).

We report all found detectors in Table 5.2 (i.e., SES detectors), Table 5.3 (i.e., MES detectors), and Table 5.4 (i.e., IS detectors). Note that, in case one detector targets multiple specification types, we select the most generic type for classification, applying the following order: SES<MES<IS. Finally, we also report the results of the evaluation of these misuse detectors in Table 5.5.

Table 5.3.: Overview of API misuse detectors with multiple usages, explicit specification (MES). Labels according to Table 5.1

Misuse Detector	L|F	A	S	C	I	P
Acharya/Xie [AX09]	C	✗	C_C	?	SCA, FPM	✗
Alattin [TX09a]	Java	✗*	C_C	$\times_A$	SCA, FPM	✓
APICAD [WZ23]	C, C++	✓	C_C, D	?	SCA, FPM, NLP	✗
CrySL [KSA+21]	Java, JCA, Android	✓	E	–	manual	✗
Doc2Spec [ZZXM09]	Java (JavaDoc)	✗	D	-	NLP, HMM	✗
EMDetect [ÇM18]	C#, Java	✗*	C_C	I	SCA, FPM	✓
Li et al. [LZT+24]	Java	✓	C_C, C_A, D	$\times_A$	SCA, FPM	✓
OCD [GS10]	Java	✗	C_C	I	DCA, FPM	✓
Pradel et al. [PJAG12]	Java	✗	C_C	$\times_A$	DCA, FPM	✓
Pradel/Gross [PG12]	Java	✗	C_C	I	DCA, FPM	✗
Ren et al. [RYX+20]	Java	✗	D	–	NLP	✗

Table 5.4.: Overview of API misuse detectors with implicit specification (IS). Labels according to Table 5.1

Misuse Detector	L|F	A	S	C	I	P
ALP [KL21]	Java	✓	C_C	$\times_A$	SCA, FPM, AL	✗
ARBITRAR [LMC+21]	C, C++	✓	C_C	?	SCA, FPM, AL	✓
F-LSTM/S-LSTM [OGKY20]	Java, JCE	✗	C_C	$\times_A$	SCA,DL	✓
Salento [MCJ17]	Java, Android	✓	C_C	×	SCA, DL	✓
StandardTrans/Target-ComTrans [YRW22]	Java, JCE	✗	C_C	$\times_A$	SCA, DL	✓

Conceptual Comparison of API Misuse Detectors Based on the selection procedure, we found 30 misuse detectors from previous work (cf. Table 5.2, Table 5.3, and Table 5.4), which were evaluated in 41 single evaluations (cf. Table 5.5).

Program Language: We found a majority of 24 of them targeting the Java and/or Java frameworks, such as Android or the Java Cryptography Extension (JCE). Six detectors target C or C++, and a single detector (i.e., EMDetect [ÇM18]) targets C# and Java.

Benchmarks: When considering the 41 evaluations, we found that a majority of 27 evaluations used existing benchmark datasets, namely:

- eleven of them applied MUBench [ANN+16]
- six of them a JCE-based dataset from OuYang et al. [OGKY20]
- three of them used the DeCapo benchmark [BGH+06]
- three of them used APIMU4C [GWL+19]
- two of them used AU500 [KL21]
- one used AndroZoo [ABKLT16]
- one used a sample of 20 projects from a collection by Gruska et al. [GWZ10]

The other 14 evaluations used benchmarks that were not directly named. Note that multiple evaluations could stem from a single publication. Apart from APIMU4C, all other benchmarks in this set also targeted the Java. Nevertheless, even though the same benchmarks were applied for different evaluations, we observed that the number of applied projects (cf. column #Projects in Table 5.5) and the absolute number of ground truth positives (i.e., **#TP+#FN**) differed. Thus, the results did not ground on the same data.

This variance in the datasets may occur due to the evolution of the dataset itself. For instance, the DeCapo dataset is still actively maintained, having its current version released on December 20th, 2024[3], or MUBench, which extended the dataset after initial publications [ANN+19b]. Another reason for variances among used datasets and projects within the evaluations is that certain detectors require specific input data. For instance, APDetect [WXQ23] used only those entries from MUBench, which were compilable. Due to the variance among the experiments, we suggest that for a valid comparison, *a dedicated experiment controlling the ground truth (i.e., the dataset) is necessary.*

Insight D-1 (RQ D-L): Non-uniform Benchmarks and Comparison

Based on the analyzed misuse detectors and their evaluation, we found a variety of benchmarks with API misuses applied to evaluate misuse detectors. However, different evaluations not only applied different benchmarks but also varied the benchmark itself. Thus, evaluation results of API misuse detectors were not directly comparable.

Availability: Based on this insight, we require an *available replication package* for the misuse detectors (cf. column **A** in Table 5.2, Table 5.3, and Table 5.4). However, we found this available replication package only for 13 of the 30 detectors (≈ 43.3%). This is comparable to what we observed in the software engineering research community [HNKO20], but not preferable, since almost all software artifacts should be available for replicability. For six publications, we found links to an artifact, however, we failed to retrieve the

[3] as of 2025/01/21 denoted at https://www.dacapobench.org/

artifact from this given URL (i.e., marked as ✗*). [HNKO20]. In the appendix (cf. Section A.2.1, Table A.4), we list the retrieved URLs of all misuse detectors for replicability of our results.

Insight D-2 (RQ D-L): Non-available Replication Packages

We found that a majority of 56.7% *(17/30) of API misuse detectors in our analyzed set were not available through a replication package, or their respective links decayed. Thus, we conclude that a majority of API misuse detectors cannot be applied to other benchmarks to form comparable evaluations.*

Source of Specification Inference: Regarding the *source of specification inference* (i.e., column **S** in Tables 5.2, 5.3, and 5.4), we found that 27 of 30 misuse detectors (90%) used API client code (i.e., C_C). This observation was also in alignment with the work by Robillard et al. [RBK+13] on general API pattern inference techniques, in which the authors found 34 of the 49 analyzed miners using API client code as a source for pattern inference. Other sources used by detectors are the API documentation (D), for instance, by Ren et al. [RYX+20], who used solely the documentation to construct a knowledge graph as specification, as well as APICAD [WZ23], which augmented certain rules, mined from API client code. CL-Detector [ZCSZ21] additionally incorporated the API code (i.e., C_A) next to the client code (C_C) and enhanced their specifications. Li et al. [LZT+24] combined all three sources, namely, client code (C_C), API code (C_A), and documentation (D), to extend API usage patterns as multiple alternative constraint graphs. Another resource was the change information of client code (C_Δ), which was used by CPAM [LCP+21] to identify change patterns from historical API misuse fixes. Based on these patterns, the authors derived five specific patterns as specifications, mainly denoting the update of API method calls. Finally, additional external sources (E) were used, for instance, in CrySL [KSA+21], a specification language particularly targeted for the Java Cryptography Architecture (JCA), cryptography experts specified constraints on the JCA API manually.

Collecting Client Code: For all 27 misuse detectors using client code, we investigated *how these detectors collected the client code.* As shown in Table 5.1, we distinguished between code used in an intra-project setting (I) and code used in a cross-project setting ($\times$). In the intra-project setting, the detectors use the code for specification inference that also contains the misuse. This way, a potential correct usage is expected in other parts of the same project. In the cross-project setting, the code for inference stems from other projects and, thus, leverages potential correct usages found externally. In case the search process for other projects required some additional API information (e.g., using the API class or method names for code search), we denoted this $\times_A$. If other information was used (e.g., quality metrics on projects), we denoteed this $\times_O$. Among the 27 misuse detectors, we found eight of them using solely intra-project settings, twelve using a cross-project setting, and a single one (i.e., MUDetect [ANN+19b]) applying both settings. From those 13 detectors using the cross-project setting (i.e., including MUDetect), nine of them used API information, two of them used other information, and two others (i.e., APP-Miner [JWL+24] and Salento [MCJ17]) provided no specific selection criteria. For six detectors, we could not clearly state, based on the description, whether they applied intra- or cross-project setting (i.e., marked as ?).

Inference Techniques: Regarding the *techniques applied during inference* (**I** in Table 5.2, Table 5.3, and Table 5.4), we found that a majority of techniques (i.e., 20 of 30) applied a mixture of SCA and FPM. Particularly, SES detectors (i.e., 13 of 14) applied

this mixture. Note that we interpreted FPM very inclusively. For instance, if a technique constructed a specification and conducted a frequency-based selection, we denoted it as an FPM technique even though it did not directly apply the techniques introduced in Section 3.5. Some detectors applied techniques in addition to SCA and FPM. For instance, APICAD [WZ23] used NLP to infer specifications from the documentation, and ALP [KL21] and ARBITRAR [LMC+21] used AL. SCA usually encompassed different variants such as Abstract Syntax Tree (AST) analysis (e.g., EMDetect [ÇM18]), controlflow and dataflow analysis (e.g., Jadet [WZL07]) as well as symbolic execution (e.g., APICAD). Four detectors applied DCA instead of SCA. DCA usually included instrumenting the source code and executing the code either with predefined test cases (e.g., Pradel et al. [PJAG12] as discussed in their previous work [PG09]), interacting with an application (e.g., OCD [GS10]), or running the code with test cases generated by tools such as Randoop [PE07] (e.g., Pradel/Gross [PG12]). While CrySL [KSA+21] expects the specification to be written by domain experts (i.e., in their case, cryptography experts), Ren et al. [RYX+20] used NLP techniques to obtain specifications from the API documentation. Doc2Spec [ZZXM09] applied NLP together with an HMM to build so-called 'action-resource-pairs' from textual input of documentation, based on which the specification automaton was constructed. All investigated IS detectors apply DL or AL next to SCA techniques.

Violation Detection: All detectors required a *mechanism to detect a violation*. In general, this means that the specification is violated. SES detectors denoted this as a single mismatch between the observed usage and the specification (e.g., APDetect [WXQ23]), a significant non-similarity between the usage the specification (e.g., DMMC [MBM10]), a failed code check via an SCA (e.g., CPAM [LCP+21]), or a non-satisfied logical formula (e.g., Tikanga [WZ11]). MES detectors typically leveraged a finite state automaton, which had been used to validate observed execution traces (e.g., Pradel et al. [PJAG12]). Other techniques checked whether mined association rules were contradicted (e.g., APICAD [WZ23]), failed certain code checks via SCA (e.g., CrySL [KSA+21]), SAT solvers (e.g., Ren et al. [RYX+20]), or other model checking techniques (e.g., Acharya/Xie [AX09]). Note that even though Li et al. [LZT+24] also used patterns, their violation detection strategies applied multiple, possibly alternatives and implied a misuse if *all* alternatives mismatched. Thus, we denoted it as an MES detector. IS detectors usually described a misuse as a significant statistical deviation of the expected usage based on the latent model.

Post-Processing: Finally, we checked whether *detectors applied some post-processing techniques*, namely, subsequent filter and ranking techniques. This post-processing is necessary since reporting all violations is not meaningful, as multiple alternative specifications can be applicable for an analyzed API usage [TX09a, ANN+19b]. We found that 23 out of 30 detectors (i.e., $\approx 76.7\%$) applied some sort of post-processing. As expected, all analyzed SES detectors applied post-processing since their specification only described a single API usage specification and thus could suffer from high false positive rates due to alternative patterns. Post-processing considered different metrics, typically support and confidence obtained during FPM, the size of specifications (e.g., Pradel et al. [PJAG12]), but also several advanced metrics as discussed by Amann et al. for MUDetect [ANN+19b].

Comparison of Evaluation Results of API Misuse Detectors Even though the experimental results were not directly comparable due to the variances in the ground truth, we analyzed the *precision* and *recall* as indicators of which techniques seemed most promising.

Detectors with high Precision: Regarding *precision*, the largest obtained preci-

sion values were reported by CPAM [LCP+21], Pradel/Gross [PG12], both with 100%, Acharya/Xie [AX09] with 90.4%, CrySL [KSA+21] with 86.5%, and Salento [MCJ17] with ≈ 80% precision. CPAM obtained the large precision by only analyzing four specific API misuse types, which were checked via four manually implemented SCA techniques based on the automatically inferred specifications (i.e., using change information as well as client code). Afterward, the authors manually validated the detected misuses and reported them as issues in the original project [LCP+21]. Additionally, they did not provide an experiment regarding its recall, which limits conclusions on its generalizability. The MES detector by Pradel/Gross [PG12] inferred finite state automata as specification using DCA and frequency information. The automata were further validated by failing test cases from a random test generator. Similar to CPAM, its experiments did not report the recall. Moreover, we were not able to find a replication package. The MES detector by Acharya/Xie [AX09] also used finite state automata, however, it concentrated on error checking code and did not report results on recall. CrySL is a specification language designed to manually create specifications by domain experts. In detail, this language targeted cryptographic API misuses, and thus, the language was designed with their specific characteristics in mind. Based on manually inferred specifications, they obtained a high precision by automatically validating them using specialized SCA techniques. In addition to the large precision, they obtained a large recall on a set of known cryptographic misuses. While their approach is limited to cryptographic APIs, cryptographic misuses are considered severe and prevalent [EBFK13, MNY+18, RXA+19]. Nevertheless, its downside is the manual specification inference. As the last highly precise technique, we report Salento [MCJ17], which learned a latent non-observable specification and determined misuses as statistical divergence to this specification (cf. Section 5.2.4 for details). They evaluated different percentages of the top divergent results and found that among the Top $6-7\%$, the precision and recall were both ≈ 80%. Having 100% recall, Salento obtained a precision of 75% among the Top 8%.

Detectors with high Recall: Among the detectors which reported the largest *recall*, we found CrySL [KSA+21] with 97.1% , APP-Miner [JWL+24] with 96.7%, Alattin [TX09a] with 94.9%, ARBITRAR [LMC+21] with 89.1%), FuzzyCatch [NVN20] with $73.4-82.1\%$, S-LSTM [YRW22] with $\approx 71-83\%$, CAR-Miner [TX09b] with 80%, and Salento [MCJ17] with ≈ 80%. We already discussed CrySL and Salento. APP-Miner [JWL+24] built so-called 'API path patterns,'. These path patterns represent Control Flow Graphs (CFGs) extended with data flow information together with a clever indexing and containment mechanism to speed up the FPM. While they achieved a large recall with a precision of 51.8%, it was not clearly stated in their paper how they determined the absolute number of true positives (i.e., **#TP+#FN**). Alattin [TX09a] found alternative patterns (i.e., frequent patterns among all samples that were not covered by other frequent patterns). ARBITRAR [LMC+21] used AL of boolean feature vectors using 31 hand-made features. Based on the feature vectors, misuses were detected by using the maximum discrepancy kernel density estimation as a statistical metric to denote divergence to expected features in a candidate API usage. FuzzyCatch [NVN20] specifically targeted API misuses concerning exception handling using fuzzing rules. Thus, its evaluation was restricted to exception handling misuses, and thus, the recall was restricted as well. S-LSTM [OGKY20] used a stacked long-short-term memory DL model (i.e., using stacked input data). CAR-Miner [TX09b] inferred sequence association rules while concentrating on exception handling code.

Table 5.5.: Precision and Recall Results of Misuse Detectors - Highlighted Results with Precision and Recall above 80%.

	Short Term	Dataset	#Projects	Precision			Recall			Reference to Validation
				#TP	#FP	%	#TP	#FN	%	
SES	APDetect	MUBench	31	42	33	56%	48	117	29.1%	[WXQ23]
	APISan	APISan + APIMU4C	4	70	508	12.1%	70	22	76.1%	[LMC+21]
	APP-Miner	-	4	29	27	51.8%	29	1	96.7%	[JWL+24]
	CAR-Miner	-	5	169	94	64.3%	128	32	80%	[TX09b]
	CL-Detector	MUBench	57	-	-	40-44.3%	112	111	50.2%	[ZCSZ21]
	CPAM	-	19	44	0	100%	-	-	-	[LCP+21]
	DMMC	-	1	11	8	57.9%	-	-	-	[MBM10]
		MUBench	29+30	12	-	7.5%	24	201	10.7%	[ANN+19b]
	FuzzyCatch	-	13,463	-	8	-	734	266	73.4%	[NVN20]
				-	35	-	821	179	82.1%	
	GrouMiner	-	9	24	149	13.9%	-	-	-	[NNP+09b]
		MUBench	29+30	4	-	2.6%	7	218	3.1%	[ANN+19b]
	Jadet	-	5	37	40	48.1%	-	-	-	[WZL07]
		Rand. from 6,097 proj.	20	11	39	22%	-	-	-	[GWZ10]
		MUBench	29+30	8	-	8.8%	15	210	6.7%	[ANN+19b]
	MUDetect	MUBench	29+30	30	61	33%	95	130	42.2%	[ANN+19b]
		AU500	16	34	89	27.6%	34	81	29.6%	[KL21]
		JCE-based	-	-	-	≈31%	-	-	≈38%	[YRW22]
	PR-Miner	-	3	23	75	23.5%	-	-	-	[LZ05]
	SpecCheck	DeCapo	14	13	11	54.2%	-	-	-	[NK11]
	Tikanga	-	6	48	73	39.7%	-	-	-	[WZ11]
		MUBench	29+30	7	-	8.2%	17	208	7.6%	[ANN+19b]
MES	Acharya/Xie	-	3	264	28	90.4%	-	-	-	[AX09]
	Alattin	-	6	149	245	37.8%	149	8	94.9%	[TX09a]
	APICAD	APIMU4C	3	66	87	43.1%	48	117	29.1%	[WZ23]
	CrySL	AndroZoo	10,000	135	21	86.5%	135	4	97.1%	[KSA+21]
	Doc2Spec	-	5	100	283	26.1%	-	-	-	[ZZXM09]
	EMDetect	MUBench	4	15	145	9.4%	13	25	34.2%	[ÇM18]
	Li et al.	MUBench	-	117	45	72.2%	117	155	43%	[LZT+24]
	OCD	-	2	9	6	60%	-	-	-	[GS10]
	Pradel et al.	DeCapo	12	41	40	50.6%	35	15	70%	[PJAG12]
	Pradel/Gross	DeCapo	10	54	0	100%	-	-	-	[PG12]
	Ren et al.	MUBench	69	68	45	60.2%	68	171	28.5%	[RYX+20]
IS	ALP	MUBench	-	72	92	43.9%	117	91	56.3%	[KL21]
		AU500	16	63	78	44.7%	63	52	54.8%	
	ARBITRAR	APISan + APIMU4C	4	82	87	48.5%	82	10	89.1%	[LMC+21]
	F-LSTM	JCE-based	-	-	-	≈7-8.4%	-	-	≈67-81%	[OGKY20]
	S-LSTM	JCE-based	-	-	-	≈8.8-9.8%	-	-	≈71-83%	
		JCE-based	-	-	-	≈27-28%	-	-	≈34-37%	[YRW22]
	Salento	-	250	-	-	≈80%	-	-	≈80%	[MCJ17]
	StandardTrans	JCE-based	-	-	-	≈37-39%	-	-	≈47-48%	[YRW22]
	Target-Com-Trans	JCE-based	-	-	-	≈38-41%	-	-	≈48-49%	

Best Performing Detectors: Overall, CrySL and Salento obtained the best results regarding precision and recall. MUDetect had been a stable detector with three similar results from different research groups (i.e., disjunctive set of authors) and different datasets. In contrast, Jadet's precision variance among the three reported experiments was much larger. IS-based detectors usually obtained the largest recall values, while MES-based detectors obtained the largest precision results.

5.2.3. Limitations of Collecting Client Code for API Misuse Detectors

When considering the state-of-the-art of API misuse detectors, we found that many detectors solely relied on client code as source data (i.e., C_C). However, they provided little insight into how to obtain this client code in a realistic scenario. While we considered the intra-project setting (i.e., I) as applicable in practice, it is limited in its ability to obtain high recall since we cannot expect to find solutions for all API misuses in single software projects. In detail, we observed that detectors in an intra-project setting obtained recall values between 3.1%-34.2% (i.e., DMMC [MBM10], GrouMiner [NNP+09b], Jadet [WZL07], Tikanga [WZ11], and EMDetect [ÇM18]). In contrast, the cross-project setting (i.e., $\times$) typically applied code search techniques, which we found hardly applicable for practical misuse detection. For instance, using API-specific information (i.e., $\times_A$) requires sufficient knowledge of which API (e.g., described by class or method name) is misused. This knowledge is typically not known beforehand in practice. For instance, Oliveira et al. [OLR+18] observed this situation in the security domain, and Zhang et al. [ZUR+18] indicated this by the prevalence of API misuses in code examples from discussion forums. This way, we expect that misuse detectors collect a set of specifications for different APIs, which are then tested among all potential locations in client code that uses this API. We see some particular issues regarding the scalability of this technique:

1. It requires managing many potential specifications from many different APIs.
2. The testing of all potential API locations causes a huge overhead, particularly when the code base becomes larger and incorporates many different APIs.

Even though training of statistical models (e.g., neural networks) as used by IS detectors represents a technique to cope with the first issue, it would require frequent re-training since, due to API evolution, different versions of specifications exist and change over time [LGS21, BR22]. Therefore, we state a *limited practical applicability of API client code collection*:

Insight D-3 (RQ D-L): Limited Practical Applicability to Collect Client Code

State-of-the-art API misuse detectors are restricted regarding a practical mechanism to collect API client code based on which specifications are inferred. Since a majority of 90% *(27/30) of the analyzed API misuse detectors used client code and* 76.7% *(23/30) used client code as a single source, a majority of misuse detectors have restricted practical applicability.*

Moreover, we found that detectors make only limited use of additional information independent of client code. Particularly, we have seen that other additional information (i.e., $\times_O$) was rarely used (i.e., in two out of 30 cases) and, if so, very restrictive. FuzzyCatch [NVN20] only focused on exception handling code, which limited its generalizability. CPAM [LCP+21] used past code changes, however, it limited the detection to simple

method replacements. We assume that using such information can boost the applicability of misuse detectors and obtain a large precision.

Insight D-4 (RQ D-L): Restricted Usage of Additional Information for Finding External Client Code

Only two out of the 30 analyzed state-of-the-art API misuse detectors leveraged additional information apart from the misused API and those only in a restricted manner. Thus, there exist potential improvements in using further information sources to collect client code.

When coping with these two limitations (i.e., **Insight D-3** and **Insight D-4**) in a new misuse detector, this detector has to deal with the commonly experienced issue of low precision, particularly a large false positive rate [LW09, RBK+13, Ama18]. Since we found some detectors with reported large precision (cf. Table 5.5), whose results are hardly comparable (cf. **Insight D-1**), we further discuss their applicability for controlled experiments subsequently.

5.2.4. Selection of Comparable API Misuse Detectors

For controlled experiments, we require a selection of detectors, which we apply to known benchmarks. Therefore, we select misuse detectors for comparison, namely, to test our search and filter strategies as well as comparing to own misuse detector (cf. Chapter 6). To keep replication effort manageable, we formulate the following selection conditions:

a) The misuse detector has a reusable replication package to avoid re-implementing a technique with unknown optimized internals.

b) Specification inference targets the Java programming language APIs since it is the most observed target programming language among our analyzed misuse detectors, and there exist more ground truth datasets for Java API misuses as benchmarks.

c) Specification inference effort should be manageable, particularly avoiding necessary large training datasets or extensive manual effort, allowing efficient evaluation.

d) The misuse detector should have promising results (i.e., stable and reliable precision and recall values), and the detection mechanism should have the potential to be applicable in practical scenarios (e.g., avoiding techniques targeting a specific limited set of API misuses).

Due to condition a), we only analyzed those 13 misuse detectors that provides a replication package (cf. Table 5.2, Table 5.3, and Table 5.4). From those, we omitted the four detectors APISan [YMS+16], APP-Miner [JWL+24], APICAD [WZ23], and ARBITRAR [LMC+21], which do not target Java (i.e., condition b)). We briefly introduce the nine remaining misuse detectors in alphabetic order and analyze the other two conditions.

ALP (**A**ctively **L**earned **P**atterns) is a misuse detector developed by Kang and Lo [KL21]. ALP is based on the API Usage Graph (AUG) extension EAUG explained in Section 3.2.2. Its idea is to find discriminative characteristic subgraphs in the EAUG that indicate whether the usage is a *misuse* or *correct usage*. For this purpose, Kang and Lo collected API usages

from GitHub repositories using a modified version of the AUSearch [ATLJ20] technique, whose search queries contain the API class and method (e.g., `java.util.Iterator#next()`). Afterward, they mined frequent subgraphs using the gSpan algorithm [YH02] and manually labeled a subset of these subgraphs as correct or misused in an AL process. Based on these manual labels, a statistical test of significance, as well as the CORK scoring mechanism [TCG+09], was used to find discriminative subgraphs between *correct usages* and *misuses*. In detail, the authors used a vector representation to train a classifier. Then, the classifier was applied as a misuse detector. ALP is provided via GitHub[4] together with a code search tool to obtain API usages[5] and the modified gSpan miner[6] and thus allows replication and reuse.

In an attempt to reuse ALP, we extended this replication package to remove tight bindings to their specific test dataset. However, we found, while experimenting with ALP, that it typically required a large number of API usage examples for each single pair of an API class and method. Otherwise, too few training samples did not yield statistically significant discriminative subgraphs. A large number of samples required large training datasets for single class-method pairs (i.e., Kang and Lo [KL21] collected $\approx 2,000$ samples per pair) together with manual labeling sessions for each training set. Thus, we decided to exclude ALP for comparison due to condition c).

CL-Detector [ZCSZ21] is a misuse detector combining knowledge from API usage and API implementation. First, it mines AUGs from API client code using the same Frequent Graph Mining (FGM) technique as MUDetect, which they refer to as *C-extraction*. In their experiments, the authors obtained the client code via the Boa framework [DNRN13]. Then, they extended the mined patterns by analyzing their referred API methods and classes (i.e., used or overridden) in the API code (i.e., the library). Particularly, they applied eight different strategies to extract constraints describing conditions, exception handling, and order relations in API code. They analyzed `throw`- and `assert`-statements as well as JavaDoc statements indicating required null-checks. This way, they also obtained AUGs from the API code. They denoted this step as *L-extraction*. Finally, they combined both AUGs by merging overlapping and overriding conditions favoring those from the library. For misuse detection of a particular API usage, they matched its related AUG to the AUG patterns and mark potential violations as misuse if no alternative matching pattern exists. Similar to MUDetect, they applied a ranking of the violations. Their misuse detector is available on GitHub[7].

Even though CL-Detector requires the presence of the API code, most steps are automated, and thus, we considered it as a possible candidate for comparison. However, when replicating the results, we found that the software artifact misses the component to combine both results from C- and L-extraction. We contacted the authors[8] but did not get a reply regarding this issue. Thus, we could not apply CL-Detector to a new dataset without implementing this technique on our own and, therefore, excluded it due to condition a).

[4] https://github.com/ALP-active-miner/ALP last accessed: 2023/11/17
[5] https://github.com/kanghj/github-code-search last accessed: 2023/11/17
[6] https://github.com/kanghj/gspan_cork last accessed: 2023/11/17
[7] https://github.com/subZHS/CL-Detector last accessed: 2024/03/21
[8] We opened an issue in the respective repository https://github.com/subZHS/CL-Detector/issues/1 (last accessed: 2025/01/21) and contacted all authors via email

CPAM [LCP+21] has the notion of reusing already fixed API misuses based on a Version Control System (VCS). In detail, Liu et al. analyzed 1,162 open source projects from GitHub and filtered code changes based on their respective commit messages (i.e., containing fix-indicating keywords) and metadata (e.g., size or whether code was solely added or deleted). They used an AST diff tool to determine modified arguments in API method calls, replaced API method calls (either from the same or a different class), added checks to API calls, or changed receiver objects. They observed the frequency of certain API changes by weighting between in-project and cross-project frequency. Based on their observed frequency results, they implemented four SCA checkers to detect misuses. They provided their results on their project webpage[9] and the source code for specification inference on Github[10]. However, they did not provide their four SCA techniques to detect API misuses.

Thus, we excluded CPAM due to condition a).

CrySL [KSA+21] is a specification language for cryptography APIs by which cryptography experts can easily formulate specifications. Then, an SCA technique automatically validates these specifications. In detail, CrySL intends to verbalize only a small set of correct API usages (i.e., whitelisting) and defines different kinds of characteristics (e.g., dataflow or typestates) typically as regular expressions. For misuse detection, a specification written in CrySL is translated into an SCA technique targeting control and data flow properties. The static analysis is conducted on single object traces (i.e., statically obtained method calls on a single object), for instance, by checking forbidden method calls, as well as the interaction of object traces, such as required method sequences (e.g., initializations). The language has been developed together with cryptography experts and has been shown to be applicable to the Java Cryptography Architecture (JCA) using Android apps from the AndroZoo benchmark [ABKLT16]. CrySL is available on GitHub[11].

CrySL and its already provided specifications are limited to cryptographic APIs, and extending the specification to other APIs requires additional manual effort as well as specific domain knowledge. Moreover, CrySL is designed with cryptographic APIs in mind and, thus, can not be directly generalized to other APIs, for which whitelisting is not applicable. Thus, we exclude CrySL due to condition c) and partially due to condition d).

FuzzyCatch [NVN20] is based on Android apps and applies fuzzy rule inference by using Graph-based Object Usage Models (Groums) [NNP+09b]. In detail, Nguyen et al. particularly analyze exception handling code by fuzzy rules representing which API method calls are correlated with certain exception types and which exception types are correlated with certain exception handling code. The fuzzy rules contain a probability representing how frequently a rule is found applicable in the training data. By defining three different kinds of risks (i.e., thresholds of probabilities), they used different sets of fuzzy rules and denoted a violation (i.e., missing calls) as misuses.

FuzzyCatch is restricted to exception handling code, and thus, we excluded it based on condition d).

Jadet [WZL07] specifies API usages by formulating temporal rules on method call orders from a statically obtained object usage model (i.e., finite state automata with edges labeled

[9] https://cpam2019.wixsite.com/mysite last accessed: 2024/03/21
[10] https://github.com/APIMisuse/ApiChangePattern/ last accessed: 2024/03/21
[11] https://github.com/CROSSINGTUD/CryptoAnalysis and https://github.com/CROSSINGTUD/Crypto-API-Rules both last accessed: 2024/03/21

by method calls). Patterns represent sets of ordered method call pairs, which are obtained using FPM on pairs (i.e., as items). By ranking patterns using the confidence metric and applying formal concept analysis, Jadet detects violations as missing calls in the expected orders. Jadet is available on the institutional webpage[12].

When we analyzed the evaluations of Jadet in Table 5.5, we found that Jadet tended to have a low precision. This result can be reasonable since it only considers simple method pairs, which easily can mark many false positive results. Thus, we excluded Jadet due to condition d).

MUDetect is a misuse detector developed by Amann et al. [Ama18, ANN+19b]. It infers frequent subgraphs among a set of API usages represented by a set of AUGs, the data structure introduced by Amann et al. themselves (cf. Section 3.2.2). MUDetect applies an FGM approach based on the Apriori algorithm (cf. Section 3.5.2) to infer these patterns. In detail, the authors targeted the problem of the exponential increase of pattern candidates in graph mining by extending only by adjacent nodes, which can be found in example AUGs. Moreover, they clustered equivalent pattern candidate extensions by identifying isomorphic graphs among them. This identification was done by a heuristic graph vectorization named *Exas vectors* by Nguyen et al. [NNP+09a]. They conducted a ranking of different pattern candidates using different interestingness measurements (cf. Section 3.5.3). Finally, they detected violations as partial overlaps to candidate AUGs. They applied further ranking strategies of violations to handle alternative patterns. They experimented with different settings. First, they controlled the donor code either in *per-project* (i.e., I) or *cross-project* ($\times_A$) setting. Second, they controlled the count value for support on the considered scope, namely, how often a pattern occurs in a single scope. They distinguished between *within-method*, *cross-method*, and *cross-project*. MUDetect is available on GitHub[13], which allows the reuse of this software artifact.

MUDetect is a stable technique, as indicated in Table 5.5. We found its replication package reusable with a manageable effort. Moreover, we denote it a plausible technique in realistic scenario. Thus, we used MUDetect in our experiments.

Li et al. The misuse detector by Li et al. [LZT+24] is a recent technique at the time of writing this thesis[14]. It leverages multiple sources to infer specifications, namely, *API usages from client code*, together with information on *the library code* and *documentation* of the API. In detail, the authors converted client code into AUGs (cf. Section 3.2.2) and constructed so-called '*API usage constraints*.' These constraints were subsequently filtered (among others by frequency). Such constraints were also extracted from the library code and the documentation of the API. All constraints were related to previously selected APIs. This way, the authors merged different constraints based on the API as so-called '*API constraint graphs*.' In detail, they had different heuristics to resolve conflicts among contradicting constraints (e.g., between those from documentation and library code) as well as to construct *alternative* API constraint graphs. The detection was conducted by parsing client code as AST and checking whether at least one of the alternative constraint graphs is satisfied. Otherwise, the API usage was considered as a misuse. The misuse detector is available on figshare[15].

[12] https://www.st.cs.uni-saarland.de/models/jadet/JADET.zip last accessed: 2024/03/21
[13] https://github.com/stg-tud/MUDetect last accessed: 2023/11/17
[14] Note that this work was found in the repeated forward snowballing conducted on 2025/01/29
[15] https://doi.org/10.6084/m9.figshare.24552193 last accessed: 2025/01/31

Even though it achieved good precision (i.e., Table 5.5), we found some essential work to apply this technique to our own experimental setting. For instance, finding library code and documentation to APIs is currently not covered, and code sections in their artifact have to be customized (i.e., configure necessary file paths within the source code). Therefore, at the time of writing, the effort for specification inference was considered to be too high (i.e., condition c)). Nevertheless, we recommend a comparison to this technique in subsequent research.

Salento [MCJ17] infers a latent non-observable specification using Bayesian learning over code features and observed behavior (i.e., method call sequences). Particularly, they train a topic-specific recurrent neural network, which distinguishes between different APIs and their specific usage contexts. This network produces a stochastic distribution of an expected behavior based on the code features of a candidate program. This expected behavior is then compared to a statically obtained behavior (i.e., also as stochastic distribution) using the Kullback-Leibler divergence. Their replication package is available on GitHub[16].

Since it trains a neural network, it typically requires a large amount of source code samples for certain APIs. Moreover, their instrumentation to obtain this training data focuses on the Android framework. Therefore, we assessed the effort to apply this technique efficiently as too high and, thus, excluded Salento due to condition d).

Insight D-5 (RQ D-L): Limited Applicability of API Misuse Detectors for Comparison

Many API misuse detectors were not directly applicable for comparison, mostly due to the non-availability of a replication package (i.e., 17 from 30 analyzed detectors). From the remaining 13 detectors, we excluded four due to their targeted programming language, four due to less promising results, two regarding the non-availability of mandatory components in their replication package, and two due to their effort to infer specifications. We only selected MUDetect [Ama18, ANN+19b] as a comparable API misuse detector.

5.2.5. Threats to Validity

Finally, we discuss potential threats to the *internal* and *external* validity of the analyzed limitations, as discussed in Section 5.1.

Internal Validity When analyzing the state-of-the-art API misuse detection, we applied an SLR process using forward snowballing [KC07] with the search engine Google Scholar. This search engine may return different search results depending on the the search time, and thus, a replication may result in a different set of publications [KLvN+20]. Moreover, the selection was partially done by the main author using the publications' title, which implies a selection bias.

Similarly, the suggested classification according to the specification type is subjective and only serves as an element to structure the discussion.

Finally, we presented the obtained performance (i.e., precision and recall) of the misuse detectors as they were presented in the respective papers. Thus, potential threats to validity regarding these experiments have to be respected as well.

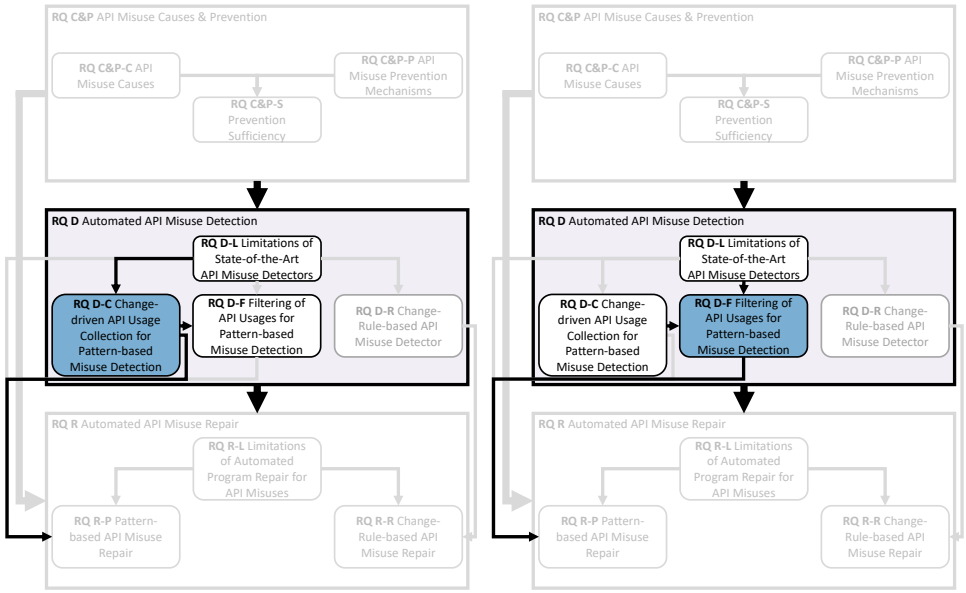
[16] https://github.com/trishullab/salento last accessed: 2024/03/25

External Validity As previously described, the classification is by no means general and could not apply to other not-considered or future misuse detectors. For instance, other classifications can further differentiate between the application domain (e.g., cryptographic APIs) or the targeted misuse type (e.g., exception handling or resource management issues).

Even though related, our results only apply to API misuse detectors but not to the more general topic of API specification inference. This domain also discusses other techniques that are intended to support other use cases, such as automated documentation generation. A detailed view of API specification inference can be obtained from Robillard et al. [RBK+13] and subsequent works.

5.3. Improving Data Collection for API Pattern Inference by Change-Based Information

This section mainly refers to our work on change-based misuse detection published in our previous work [NHO18, NHSO21]. It targets research questions **RQ D-C** and **RQ D-F**.



5.3.1. Insufficient Data Selection for API Specification Mining

We discussed in Section 5.2 that many state-of-the-art API misuse detectors relied on the principle of FPM to retrieve common API usage patterns, which served as specifications of *correct* API usage (i.e., API specification mining). Most of these pattern-based detectors produced a large number of false positives, namely, patterns that falsely blamed a correct API usage as misuse. Having too many of those "false alarms," developers would likely ignore warnings from such a system, as seen in approaches of SCA [CM04, JSMHB13, SAE+18]. Amann [ANN+19a] identified several root causes for false positives (e.g., infrequent but correct usages, missing information in the mined data structures) and reduced the false positive rate by developing a graph-based data structure (i.e., AUG) as well as validating several ranking and violation detection techniques. While achieving a significant improvement over the state-of-the-art, the obtained precision (between 21.9% and 34.1% for different settings) is still far from practical applicability (i.e., a larger proportion of false

positives) [JSMHB13].

Thus, we analyzed the main steps of typical API specification miners to hypothesize potential improvements [NHSO21]. According to the previous work discussed in Section 5.2, we derived the following five common steps for API specification mining with subsequent misuse detection:

1. Obtain a sufficient API usage example database for FPM
2. Transform API usages into a dedicated data structure (e.g., execution traces [YEB+06], syntax trees [AS14], AUGs [Ama18])
3. Retrieve a set of frequent API usage patterns by a form of FPM (e.g., association rule mining [LZ05], Frequent Sequence Mining (FSM) [ZXZ+09], FGM [Ama18])
4. Filter and rank patterns based on some kind of interestingness measurements [LL15]
5. Compare the interesting patterns to the API usage and mark deviations as API misuses [Ama18]

In recent work, the last four steps received more attention than the first one, namely, obtaining an API usage database to mine from. Le Goues et al. [LW12] analyzed the impact of code quality metrics as a pre-processing step of usage examples in the database. Zhong et al. [ZXZ+09] proposed a clustering technique of API call sequences and conducted FSM on single clusters instead of the complete input space. During evaluation of API specification mining, usage examples for the database are usually obtained from 1) the same project that contains the misuse [Ama18, ANN+19a, WZ23], 2) a set of projects collected due to some project characteristics [YMS+16, ÇM18, ZCSZ21, JWL+24], and 3) specific code examples exactly containing the type and/or the API element (e.g., method or field) of the misuse [TX09b, TX09a].

The first variant may result in no usage pattern at all, particularly if the API has been (mis)used the first time in the project since no correct usage pattern of that API exists. In the second case, specification mining may result in many unrelated patterns since the misuse and usage examples do not necessarily relate to each other. Note that this also occurs in the first case since usage patterns from other internally used APIs are mined as well. In contrast, the last variant directly refers to the misuse at hand, however, this does not work in practice. While it is sufficient for validating and comparing different API specification miners, developers trying to identify latent misuses in their code usually do not know if and which API elements are misused.

Thus, current specification miners lack sufficient techniques for obtaining an API usage database. Without such a database, subsequent steps of mining usage patterns and misuse detection can be harmed. This issue is similar to issues seen with classifiers when insufficient data is used [AM18]. For API misuse detection, this would mean that a misuse cannot be detected since no pattern is found or many false positive patterns are found, increasing the false positive rate.

5.3.2. Concept of Change-Based Information to Collect API Usages

The basic essence of our concept for detecting API misuses is to leverage change information from a potential misuse-introducing code commit from a Version Control System (VCS), such as `git`. This change information contains not only the code difference between two

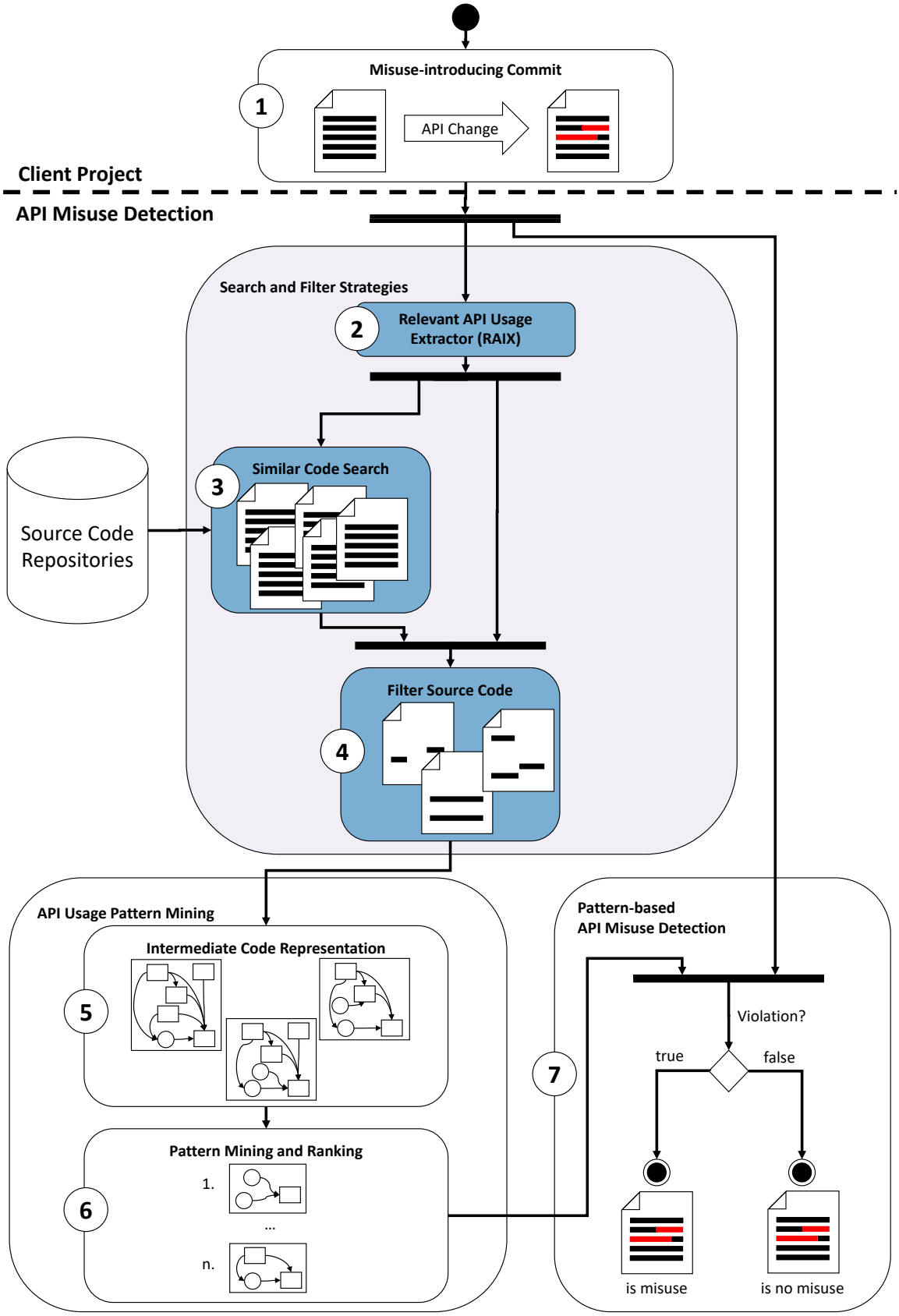


Figure 5.1.: Integration of RAIX and our search and filter strategies into a pattern-based API misuse detection process using commits (adapted from our previous work [NHSO21]).

subsequent versions but also further meta-information (e.g., author of the change, commit message), which we hypothesize to have more valuable insights for precise misuse detection. Moreover, we assume that code changes are usually much smaller than the whole code base and thus allow a more concise misuse localization (i.e., only in those code sites that were actually changed). Our main idea is to distill information on the API usage from the code change and apply this knowledge to find similar and potentially correct usages to compare them with the potential misuse [NHO18].

Particularly, this concept targets the two research questions, **RQ D-C** and **RQ D-F**.

Overall Process We extended and validated this idea [NHSO21] to fit into a realistic development scenario, as depicted in Figure 5.1. We assume a client project in which a developer commits a code change containing an API misuse to the project's repository ①. This way, the aspired misuse detection process can serve as a step within the Continuous Integration (CI) system, which refuses commits in case they contain an API misuse. The first step of this process is the API change analysis using our artifact *Relevant API Information Extractor (RAIX)* ②. It conducts a static analysis of the code change and extracts keywords and API imports, both characterizing the API usage within the commit. Both extracted pieces of information serve as input for the subsequent *search* ③ and *filter*-strategies ④, which yield a set of similar source files originating from source code repositories. Then, we apply a state-of-the-art API usage pattern (or API specification) miner by transforming the similar code samples to an intermediate representation ⑤ and apply the miner with ranking strategies of the patterns ⑥. Finally, we verify whether the changed code of the commit complies with the inferred patterns, namely, they are not violated ⑦.

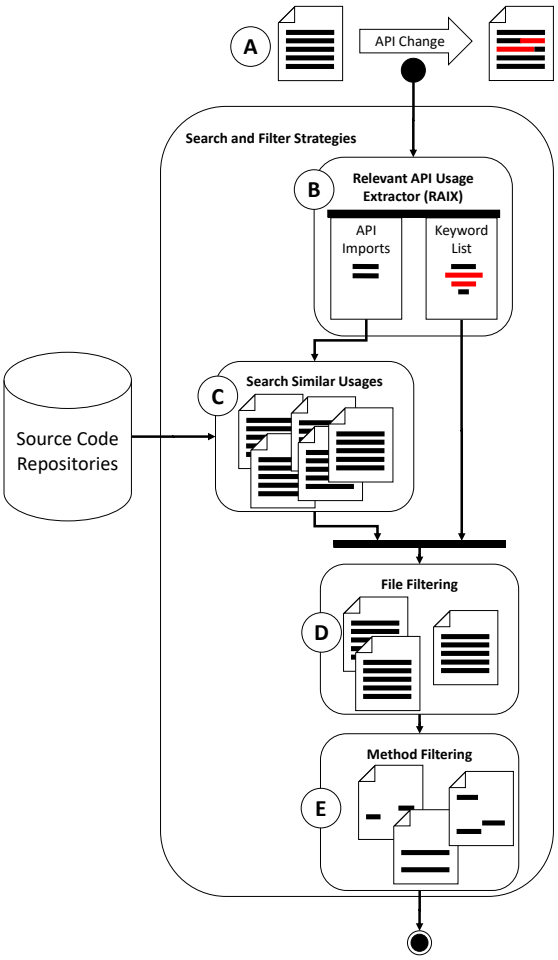


Figure 5.2.: Detailed view on RAIX and the search and filter strategies (adapted from our previous work [NHSO21]).

Search and Filter Strategies The main contribution concentrates on the *search and filter strategies* (i.e., ① - ④) and validates their impact on the pattern miner and the subsequent API misuse detection. Thus, we further zoom into this part in Figure 5.2.

API Change: First, we analyze the code change in Ⓐ by applying utilities from the VCS (i.e., `git diff`). This way, we obtain the changed code lines based on which we infer those method declarations that contain at least one changed line using SCA. Note that

we use whole method declarations instead of single code lines since they provide much more context of the underlying API usage and, thus, allow a more detailed, subsequent code search. Moreover, considering a larger context increases the chance of detecting more misuse types, for instance, if the change adds a second, redundant method call, which can represent a misuse [ANN+19a]. By applying commit-based analysis on changed method declarations, we hypothesize that these still significantly reduce the amount of code to be analyzed by the misuse detector.

API Extraction: For each changed method declaration, we extract those code elements characterizing the API usage (cf. Ⓑ). Particularly, this denotes a list of change-related API `import` statements from third-party libraries and a list of keywords describing API elements affected by the change (e.g., API type names or API method calls). The particular selection of `import` statements and keywords was motivated by reviewing real-world misuses from the MUBench benchmark [ANN+16] as well as from insights by Zhong et al. [ZXZ+09] on code features characterizing API usage. Note that we concentrate on third-party libraries due to two reasons: First, we likely will not find usages of internal APIs in external repositories, and second, language-specific elements (i.e., from `java.lang`) are far too common and would introduce too much noise for subsequent filtering.

We illustrate the API import and keyword extraction by means of an example in Figure 5.3. In this example, we assume that the method `doSomething` has been changed by a developer. We identified change-related API `import` statements through the following procedure. First, we check for each `import` statement in the main class (i.e., the only `public class` in the source file containing the changed method) whether it depicts a third-party library. Second, we analyze whether the third-party import is used in the changed method. We applied a heuristic matching the first three qualifiers of the `import` statement with those of the `package` statement of the source file. In case both match, the `import` statement is denoted as internal API and thus discarded from the API imports list. The rationale to use three qualifiers stems from the naming convention for packages in Java[17], which denotes that packages are usually identified by the companies' internet domains succeeded by the package name, resulting in at least three qualifiers. If only one or two qualifiers are used as package identifiers, we only check whether these are prefixes of the `import` statements. In case no `package` statement is given, which never occurred in our evaluation, no API imports are inferred. Within our example in Figure 5.3, the `import` statement from class `QClass` is not considered since its prefix (i.e., `my.own.pkg`) matches the prefix of the respective `package` statement.

Then, we relate `import`s to changed methods if the class imported in the `import` statement is

- used as parameter type in the method's signature
- used as a `return` type in the method's signature
- used as a `throws` type in the method's signature
- explicitly used in an expression in the method's body
- inherited from the class (i.e., `extends`) containing the method, and the method is annotated with an `@Override` statement

[17] https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html last accessed: 2023/06/06

```
1  package my.own.pkg.subpkg;
2
3  import a.b.AClass;
4  import a.b.BClass;
5  import a.b.CClass;
6  import x.y.ZClass;
7  import x.v.*;
8  import my.own.pkg.QClass;
9
10 public class Foo extends AClass {
11     @Override
12     protected ZClass doSomething(BClass bObj) {
13         super.doSomething(bObj);
14         QClass myQObj =
               callThisMethod(RClass.rMethodCall());
15         return myQObj.mergeWithZClass(bObj);
16     }
17 }
```

Code Listing (5.1) Code Sample API Extraction

API Imports

```
import a.b.AClass
import a.b.BClass
import x.y.ZClass
```

Keyword List

```
BClass
ZClass
rMethodCall
AClass
doSomething
mergeWithZClass
callThisMethod
```

Figure 5.3.: Example of a keyword extraction for the `doSomething`-method. Left Code Listing 5.1: Source code, from which keywords are extracted. Right: Lists of extracted API `import` statements and keywords (adapted from our previous work [NHSO21])

Considering our example in Figure 5.3, we extract the classes `AClass`, `BClass`, and `ZClass` while ignoring class `CClass` since it is not used in the `doSomething`-method according to our definition. Note that we ignore classes imported via wildcard syntax (e.g., class `RClass` through `import x.v.*;` used in line 14) since this would add too many internal classes, producing too much noise and perturbing the code search.

Regarding the keyword list, we added all class names referenced in the change-related API `import` statements (i.e., `AClass`, `BClass`, and `ZClass` in our example). Moreover, we add all method calls (i.e., including also internal and `java.lang` method calls) to the keyword list (i.e., `doSomething`, `callThisMethod`, `rMethodCall`, and `mergeWithZClass` in our example). This procedure creates a long keyword list, which we compensate for by only applying the keyword list in the filter step and using different proportions of keywords for filtering (cf. subsequent discussion on the *satisfaction ratio*). Finally, in case the changed method overrides (i.e., annotated by `@Override`) a third-party method, we add its method name to the keyword list (i.e., `doSomething` in our example). The rationale is that many third-party APIs are used via inheritance and do not necessarily call this method. Note that we avoid adding duplicates in both lists (i.e., API imports and keywords). This procedure is automated in our software artifact named RAIX.

Search Strategies: After producing both lists, we use them to conduct the code search by implementing two search strategies, namely, $search_{loc}$ and $search_{imp}$, in step Ⓒ. The first strategy (i.e., $search_{loc}$) determines the *location* of the code search. We distinguish between *internal* (i.e., within the same project containing the potential misuse) and *external* (i.e., in other projects). The second strategy (i.e., $search_{imp}$) varies, which exact `import`-statements are used, namely *all imports* (i.e., all `import`-statements from the API imports list) or only *misused imports* (i.e., `import`-statements importing the misused API classes). Note that both strategies only use the API imports list. We analyzed $search_{loc}$ since Amann stated that API usage patterns could be retrieved from the same as well as from

foreign projects [Ama18]. They found that pattern mining-based misuse detection yields better results with patterns inferred from foreign code. However, the domain of automated program repair relies on the *plastic surgery hypothesis*, namely, that patches can often be found within the same project as the bug [LGPR19]. Thus, we evaluate the effect of both variants. The idea behind the strategies in $search_{imp}$ is whether it is worth conducting a pre-search analysis to determine the misused API. On the one hand, we can assume that searching only with the misused API(s) yields more related API usages since it is not perturbed by "noisy" `import` statements. On the other hand, we argue that having a larger number of `import` statements more precisely describes the context of the API usage together with other APIs and thus yields fewer but better-matching usage examples.

Filter Strategies: With the previous step, we obtained a set of source code files, which now will be further filtered. First, we apply a file filter ($filter_{file}$) in step (D). As stated before, the keyword list is "noisy" in the sense it can contain keywords related to internal and `java.lang` elements. Moreover, we do not expect that all external usages also contain all keywords present in the changed method. Thus, we introduce the notion of a *satisfaction ratio* (sr), which denotes the proportion of keywords from the keyword list found in the respective source file. Particularly, having the keyword list as set $kwSet$ and a source file $srcFile$, sr is defined as follows:

$$sr(srcFile, kwSet) = \frac{|\{kw \in kwSet \text{ if } srcFile \text{ contains } kw\}|}{|kwSet|}$$

Having an sr of 0 denotes that the source file contains none of the keywords in the keyword list, while an sr of 1 describes that all keywords are found in the file. We filter the source files based on a minimal sr. Since we did not expect that any of the extreme cases (i.e., 0 or 1) achieve optimal results, in our experiments, we varied the minimal sr value.

Finally, we apply a method filter ($filter_{method}$) in step (E). Its goal is to remove methods from the filtered source files that do not contain any keywords from the keyword list. The rationale is that our misuse detector represents API usages in an intraprocedural manner, and thus, we hypothesize that reducing the number of non-related methods finds more related patterns for better misuse detection. Particularly, we parse each method in the source file to token sets, filter out syntax elements and keywords from the Java programming language, and check whether at least one token from this set represents a keyword from the keyword list. Note that we do not apply the sr to methods due to two reasons. First, we test all possible combinations of the *search* and *filter* strategies, and thus, introducing further variables increases the number of configurations to test. As shown in the next section, we tested for each misuse 40 (i.e., $2 \cdot 2 \cdot 5 \cdot 2$) configurations on 37 misuses from the MUBench dataset [ANN+16], leading to currently $1,480$ different configuration runs. Second, we assess the effect of a subsequent sr filtering on the method level as hardly significant since the previous $filter_{file}$ step already filtered out non-related files, and methods are far more fine-grained.

Note that we implemented both filter strategies (i.e., $filter_{file}$ and $filter_{method}$) separately to the search step to evaluate the impact of the single steps in our experiments. In a practical setting, it is reasonable to combine search and filtering to gain efficiency.

5.4. Experimental Data and Processing

In this section, we shortly describe our dataset and experimental setting.

5.4.1. API Misuse Datasets

In our experiments, we used two datasets of known API misuses, named MUBench[18] [ANN+16] and AU500[19] [KL21]. These datasets represent a collection of real API misuses in the Java programming language obtained from different open-source projects.

We found for MUBench some biases (i.e., discussed subsequently) that could influence the results and the overall validity. Thus, we pre-processed MUBench. Initially, we obtained a set of 245 API misuses based on 63 Java open source projects[20] as well as four synthetic misuse groups (i.e., clusters of misuses written by the authors themselves) together with meta-information (e.g., VCS, misused API, fixing commit). Then, we selected only those misuses from projects applying the VCS `git`, which is one of the most frequently applied VCS[21], resulting in 103 misuses. Finally, we discarded further 66 misuses due to the following reasons:

- they represented duplicates, particularly the entries from the `jodatime` project, which contained many similar misuses from test cases, which may bias the results (36);
- they were misuses of the `java.lang`-API, which were excluded by RAIX (cf. Section 5.3.2) (18);
- they were misuse of an internal API, i.e., originating from the project containing the misuses, which were excluded by RAIX (cf. Section 5.3.2) (8);
- they were non-distinguishable misuses (i.e., misuses were equal and shared the same commit, method declaration, and class), and thus we only kept one version (2);
- they were caused by a false parameter value, and this misuse type was not handled by our method (2).

We used the remaining 37 misuses to validate the commit size (cf. Section 5.5.1), the filter techniques (cf. Section 5.5.2), and the misuse detection (cf. Section 5.5.3).

Moreover, we applied AU500 as a second dataset of 500 manually validated API usages from 16 open-source projects written in Java and all versioned with `git`. Each usage is labeled either as *misuse* (i.e., 115 in AU500) or as *correct usage* (i.e., 385 in AU500). For each usage, there exists meta-information, for instance, a link to the `git` repository, the hash of the analyzed commit, and the location of the misuse in the source code. We kept all misuses from AU500. Note that AU500 has no overlapping misuses with MUBench. We applied it to validate the misuse detection (cf. Section 5.5.3).

5.4.2. API Misuse-Introducing Commits

As denoted in the overall concept (cf. Figure 5.1), our main idea is based on the analysis of misuse-introducing commits, namely, commits whose changes made a misuse apparent.

[18] https://github.com/stg-tud/MUBench last accessed: 2019/02/05
[19] AU500 is part of the ALP replication package https://github.com/ALP-active-miner/ALP last accessed: 2023/06/23
[20] as of 2019/02/05
[21] according to Open Hub managed by Synopsys, Inc. https://openhub.net/repositories/compare last accessed: 2023/06/23

In the MUBench dataset, however, we only had the fixing commit (i.e., the change that fixes the misuse). We inferred the misuse-introducing commit by first checking the fixed version of the code, then detecting the changed lines via the `git diff` command, and manually identifying those code lines that essentially describe the API misuse. Based on these lines, we checked out the previous, misused version and ran `git blame` to identify which commit added those lines. In case multiple commits were responsible for introducing a misuse, we selected the latest one since this represents the state when all "ingredients" of the misuse were together. This procedure is essentially the `git`-adapted version of the SZZ algorithm [SZZ05], originally introduced for the CVS VCS.

The AU500 dataset consists of many more API usages and not all of them represent misuses. Thus, not all had a fixing commit, so we applied a simpler approach. Particularly, we applied keyword and API import extraction on the method in the class of commit provided by the AU500 dataset. This way, we mimic the situation that exactly this method has been changed in the previous commit.

5.4.3. Similar Source Files

In step Ⓒ in Figure 5.2, we required a database of API usages from source code repositories to conduct the $search_{loc}$ strategy (i.e., *internal* and *external* code search). For the *internal* search, we downloaded and checked out the misuse-introducing commit and collected all `*.java`-source files except the source file containing the misuse. According to the *external* search, we applied the REST API of the *Searchcode* engine[22].

We selected *Searchcode* since it searched within projects from well-known code repository platforms, such as *GitHub*, *BitBucket*, *Google Code*, or *GitLab*. Compared to other code search engines like Boa [DNRN13] and GHTorrent [Gou13], it allowed accessing and downloading individual source files from the current state of the repository. At the time of our experiments, *Searchcode* returned, due to an internal restriction, at most 1,000 source files for each search request sorted by *relevance*. According to the developer[23] of *Searchcode*, *relevance* denoted the proximity of the search terms found in the source files. For instance, a search with the keywords `foo` and `bar` ranks a source file containing the String `"foo bar"` higher than those having both keywords distributed among the complete file. This definition of *relevance* was reasonable for our use case since we conducted the search by using the `import` statements extracted before, which should usually appear closely together at the beginning of each source file.

For each misuse, we obtained similar source files by running two search runs on *Searchcode* to test the $search_{imp}$ strategy (i.e., searching only with the *misused imports* compared to searching with *all imports*). Note that we only searched with the *misused imports* strategy in case the change analysis inferred the respective `import` statement from the commit. This way, we obtained at most 2,000 source files for each misuse (1,000 for each search run). For MUBench, we downloaded the code files between February 7th and February 8th, 2019, and repeated the search due to an error for `logblock-logblock-2_15` misuse on December 12th, 2019. For the AU500 dataset, we downloaded the source files on June 15th, 2021. Since *Searchcode* accessed live code data, we stored all downloaded raw source files locally and provided them together with our replication package[24]. This way, we were able to conduct all subsequent filter strategies and mining approaches on a consistent dataset,

[22] https://searchcode.com/api last accessed: 2023/06/26
[23] we contacted him via email
[24] http://doi.org/10.5281/zenodo.15594600

which kept the bias introduced by the quality of the *Searchcode* results consistent among our analyzed configurations. We further ignored source files from the same project as the misuse by comparing the prefix of the `package` statement of the file to the misuse file at hand. This way, we also could handle source files originating from forked projects. Moreover, we excluded source files for which the generation of the intermediate source code representation (i.e., AUGs) occupied too much memory and caused our evaluation script to crash. For MUBench, we, therefore, had to exclude externally found source files for 13 misuses (i.e., for nine misuses, we excluded a single source file, and for four misuses, we excluded two up to nine source files).

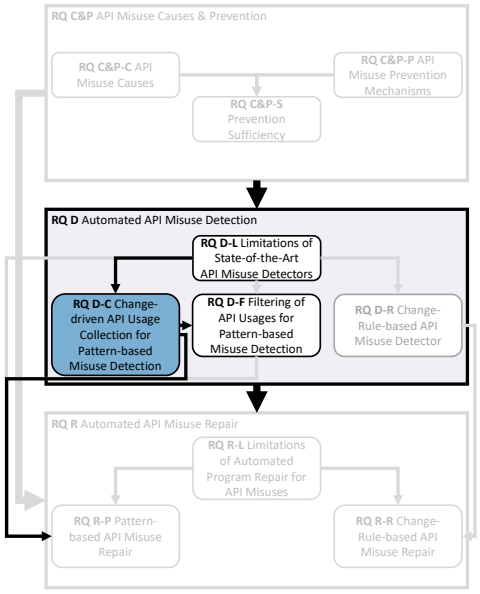
5.4.4. API Usage Graphs as Intermediate Representation

We evaluated the effect of the search and filter strategies on pattern mining and misuse detection by applying MUDetect [ANN+19a]. As intermediate representation, they used the AUG, which we already introduced in Section 3.2.2. For the purpose of our experiments, we implemented a serialization of the AUGs to store them permanently. This way, we saved some processing time for the mining by avoiding frequent re-generation of AUGs from source code.

5.5. Validation of the Impact on Change-Based Information for Pattern-Based API Misuse Detection

In this section, we present the experimental results when applying RAIX with subsequent search and filter strategies.

5.5.1. Validation of Commit Sizes



We assessed to which degree commits of API changes were a valuable source to detect API misuses, particularly whether API-specific change analysis could effectively reduce the size of the commits and, thus, the analysis effort. This way, we targeted **RQ D-C**.

Methodology We analyzed the misuse-introducing commits from the 37 misuses from the MUBench dataset [ANN+16] regarding their change size and how effectively API change analysis (i.e., step ② in Figure 5.1) could reduce this size. Particularly, we denoted as change size the *number of changed methods* per commit, the *number of extracted API imports*, and the *number of extracted keywords* (both per changed method). The number of changed methods decides on the number of separate misuse detection threads (i.e., steps ③ to ⑦ in Figure 5.1). Thus, decreasing the number of methods for misuse detection also decreases the overall static analysis time. As can be seen in Table 5.6, some misuses had the same

misuse-introducing commit. Thus, we analyzed the commit-specific size (i.e., the *number of changed methods*) based on the *31 unique commits*.

We further analyzed the number of extracted API imports and keywords. Using the API imports, we identified how many different APIs had to be considered during pattern mining. A larger number of keywords may reduce the chance of finding similar source code files in the $filter_{file}$ step, particularly if we apply a large satisfaction sr. However, a larger number of keywords can also decrease the impact of the $filter_{method}$ step (i.e., more methods are accepted) since it is more likely to match at least one of the keywords. Thus, we expect the average number of keywords and imports to be as low as possible without being 0. Otherwise, a too large number may require subsequent filtering of the most important (i.e., those that effectively find code samples) keywords and imports. In contrast, many entries with no keywords and imports might cause no search results at all.

Our replication package[24] contains a static analyzer based on the Eclipse JDT parser analyzing the misuse introducing commit and automatically obtaining the number of *all methods* present in the current code revision (i.e., after the misuse-introducing commit) and *all changed methods* together with their extracted API imports and keywords. In case we extracted at least one API import statement, this denoted that the method change contained a third-party API according to our API extraction mechanism. Note that we retrieved the number of all methods by hashing all source files (i.e., by an MD5 hash function) and counted methods only for each single hash. This way, we discarded duplicated source files from the counting work. Moreover, we avoided issues regarding name clashes (e.g., caused by overloaded methods) by assigning each method with a unique ID. We evaluated the obtained numbers based on a `Python` script within a `Juypter Notebook`.

Results Table 5.6 depicts all 37 misuses together with misuse-introducing commits. Moreover, this table contains the *number of all methods* (A) present after the commit changes, *all changed methods* through the commit (C), as well as all changed methods containing a third-party API (E). In columns A2C and C2E, we depict the reduction achieved (in percentage) by only considering the changed methods and only considering changed methods with at least one third-party API, respectively. In the last column, *API found*, we express whether the change analysis kept the method containing the misuse and correctly extracted the import statement of the misused API (i.e., after step C2E). We checked the misused API based on the description of the misuse from the MUBench dataset. Note that the A2C reduction did not discard any of the misuse containing methods due to the definition of the misuse-introducing commit.

Overall, we observed that for 31 of the analyzed 37 misuses (83.8%), we successfully obtained the misused API import statement. For the remaining ones, the extraction obtained a more specific import, namely, inherited classes, (two misuses), obtained other imports (two misuses), and obtained no imports (two misuses).

We depict the distribution of the unique misuse introducing commits among the number of changed methods after A2C reduction in Figure 5.4. We observed that a majority of 24 from 31 unique commits (77.4%) changed less than 100 methods, and 12 commits edited at most 20 methods. However, some extreme outliers, such as `bcel_101` with $2,517$ methods, persisted. After conducting C2E reduction (cf. Figure 5.5), 25 commits changed less than 100 methods and 18 edited at most 20 methods. Still, we observed single misuses with a large number of methods that need to be analyzed (e.g., `android-rcs-rcsjta_1` with 642 methods). A summary of the effect of the reduction steps is depicted in the boxplot in

Table 5.6.: List of API misuses with their misuse-introducing commits as well as their size reduction adapted from [NHSO21].

#	misuse	repository (subdomain at https://github.com)	MIC	#methods A	C	E	reduction (%) A2C	C2E	API found
1	alibaba-druid_1	/alibaba/druid.git	de13143e0	16095	12	8	99.9	33.3	✓
2	alibaba-druid_2	/alibaba/druid.git	de13143e0	16095	12	8	99.9	33.3	✓
3	android-rcs-rcsjta_1	/android-rcs/rcsjta.git	b3445d9	10817	2275	642	79.0	71.8	✓
4	androiduil_1	/nostra13/Android-Universal-Image-Loader.git	9d77de9	737	279	131	62.1	53.0	✓
5	apache-gora_56_1	/apache/gora.git	e4db20a	1565	141	57	91.0	59.6	✓
6	apache-gora_56_2	/apache/gora.git	bbad5d213	1424	39	35	97.3	10.3	✓
7	bcel_101	/apache/commons-bcel.git	d532ec1	3475	2517	269	27.6	89.3	✓
8	calligraphy_1	/chrisjenx/Calligraphy.git	1a2d0f5d	32	10	8	68.8	20.0	✓
9	calligraphy_2	/chrisjenx/Calligraphy.git	1a2d0f5d	32	10	8	68.8	20.0	✓
10	closure_2	/google/closure-compiler.git	e5d3e5e012	11135	28	14	99.7	50.0	✓
11	jodatime_269	/emopers/joda-time.git	08a925a31	4429	54	10	98.8	81.5	✓
12	jodatime_339	/emopers/joda-time.git	9b01b9e8b	9054	21	11	99.8	47.6	✗
13	jodatime_361	/emopers/joda-time.git	7fe68f297	2556	2451	519	4.1	78.8	✓
14	jodatime_362	/emopers/joda-time.git	7fe68f297	2556	2451	519	4.1	78.8	✓
15	jodatime_363	/emopers/joda-time.git	7fe68f297	2556	2451	519	4.1	78.8	✓
16	lnreadera_1	/calvinaquino/LNReader-Android.git	a514f35	3329	81	72	97.6	11.1	✓
17	lnreadera_2	/calvinaquino/LNReader-Android.git	a514f35d	3329	81	72	97.6	11.1	✓
18	logblock-logblock-2_15	/emopers/LogBlock-2.git	5ea1b0b	70	5	4	92.9	20.0	✓
19	mqtt_389	/emopers/paho.mqtt.java.git	77aa39b9	670	608	115	9.3	81.1	✓
20	mqtt_390	/emopers/paho.mqtt.java.git	f60b3721	990	59	23	94.0	61.0	✓
21	onosendai_1	/haku/Onosendai.git	Cf2de97	1618	3	2	99.8	33.3	✓
22	openiab_1	/onepf/OpenIAB.git	00e5612	957	173	100	81.9	42.2	✓
23	screen-notifications_1	/lkorth/screen-notifications.git	fa75a61f	48	21	19	56.2	9.5	✓
24	tbuktu-ntru_473	/emopers/ntru.git	4a095cc	399	13	5	96.7	61.5	✓
25	tbuktu-ntru_474	/emopers/ntru.git	e4f8688	187	8	4	95.7	50.0	✓
26	tbuktu-ntru_475	/emopers/ntru.git	8cb6471	521	41	17	92.1	58.5	✓
27	testng_16	/cbeust/testng.git	234c85874	5557	21	19	99.6	9.5	✓
28	testng_17	/cbeust/testng.git	b68cf6de8	5479	18	17	99.7	5.6	✗
29	testng_21	/cbeust/testng.git	24341340b	5432	14	11	99.7	21.4	✗
30	testng_22	/cbeust/testng.git	79cd443f	4395	4	2	99.9	50.0	✗
31	theblueallianceа_1	/Adam8234/the-blue-alliance-android.git	be7b752	1168	10	10	99.1	0.0	✓
32	thomas-s-b-visualee_29	/emopers/visualee.git	14e3f03	152	76	33	50.0	56.6	✓
33	thomas-s-b-visualee_30	/emopers/visualee.git	14e3f03b	152	76	33	50.0	56.6	✓
34	thomas-s-b-visualee_32	/emopers/visualee.git	d4dc0ba	250	1	1	99.6	0.0	✓
35	tucanmobile_1	/Tyde/TuCanMobile.git	805f770	62	11	9	82.3	18.2	✗
36	ushahidia_1	/ushahidi/Ushahidi_Android.git	db2b310	4405	63	40	98.6	36.5	✓
37	wordpressa_1	/wordpress-mobile/WordPress-Android.git	88368deadbe	5453	70	39	98.7	44.3	✗

MIC: Misuse introducing commit; A: All methods; C: Changed methods in the MIC
E: All methods from C that contain at least one external (third-party) API
A2C: Reduction from all to changed methods
C2E Reduction from changed to changed methods that contain at least one external (third-party) API

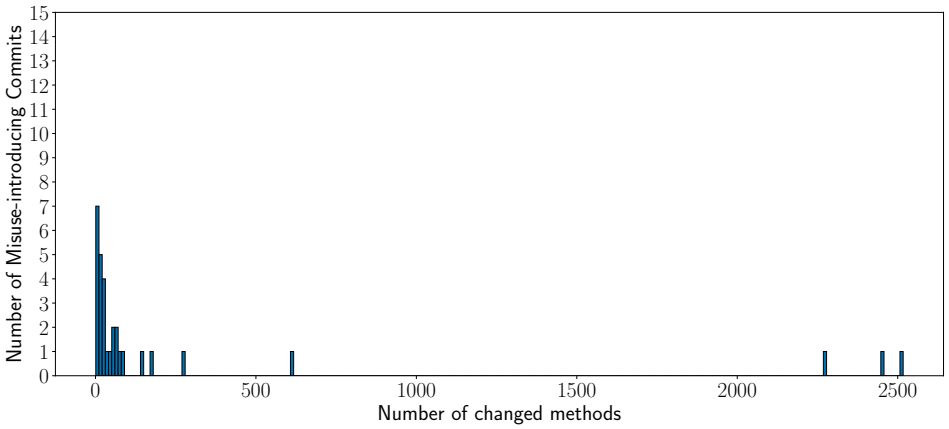


Figure 5.4.: Distribution of Misuse-introducing commits among the number of changed methods (bin size of ten) adapted from [NHSO21]

Table 5.7.: Results significant differences in the number of methods after reduction using Wilcoxon-Mann-Whitney rank sum test with Bonferroni correction (marked in gray) and its effect size using Cliff's δ.

	All Methods	**After A2C**	**After C2E**
All Methods	-	$>$ ✓0.8	$>$ ✓0.9
After A2C	$<$ ✓-0.8	-	$>$ ✓0.2
After C2E	$<$ ✓-0.9	$<$ ✓-0.2	-

Figure 5.6. While we saw a quantitative effect of considering only changed methods, we denoted a smaller effect of the C2E reduction. We tested the significance of the reduction using Wilcoxon-Mann-Whitney rank sum test [Kan06, p. 101] with $\alpha = 0.05$ and Bonferroni correction [Abd07] and measured the effect size using Cliff's δ [HK99, KMB+17]. While we determined a significant difference between all reduction steps (cf. Table 5.7), we found that based on the interpretation of Cliff's δ by Kitchenham et al. [KMB+17], the effect from A2C is *large*, while the effect from C2E is *small*. However, inspecting the C2E-step qualitatively in Table 5.6, we observe that especially misuse-introducing commits with a large number of changed methods (i.e., > 100) have a large reduction between 42.2% (i.e., `openiab_1`) and 89.3% (i.e., `bcel_101`).

Regarding the number of extracted import statements per method, we depict in Figure 5.7 the distribution of the number of statements among all misuses. Visually, we estimated that the majority had extracted at most 5-8 import statements based on the upper whisker of the boxplots (i.e., 1.5 of the interquartile range). On average, 1.8 (mean 1) methods with at least one extracted import were found, while still, some extreme outliers with 28 imports exist (e.g., `android-rcs-rcsjta_1`).

Considering the number of keywords, again, we depict the distribution of the number of keywords among the misuses as boxplots in Figure 5.8. Once again, based on the upper whiskers (i.e., 1.5 of the interquartile range), we obtained for most methods mostly 20 to 25 keywords. On average, 6.3 (mean 4) keywords are extracted from methods that used at least one third-party API (i.e., at least one API import statement is extracted). Again,

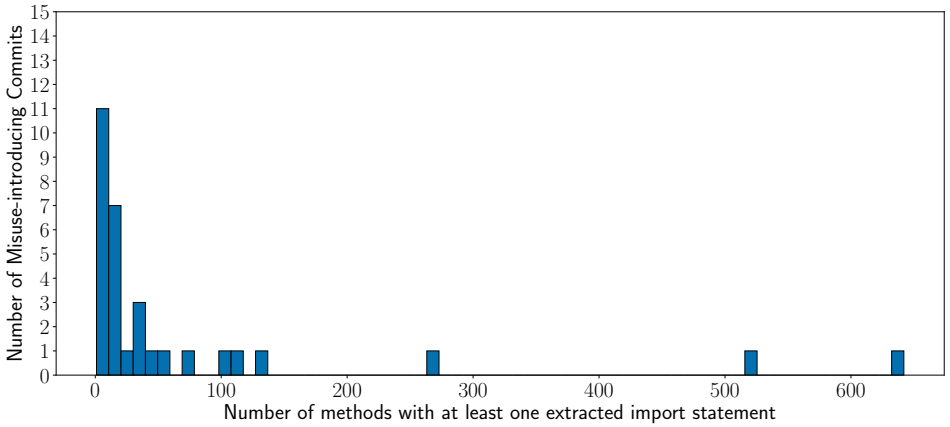


Figure 5.5.: Distribution of Misuse-introducing commits among the number of changed methods with at least one extracted external API (bin size of ten) adapted from [NHSO21]

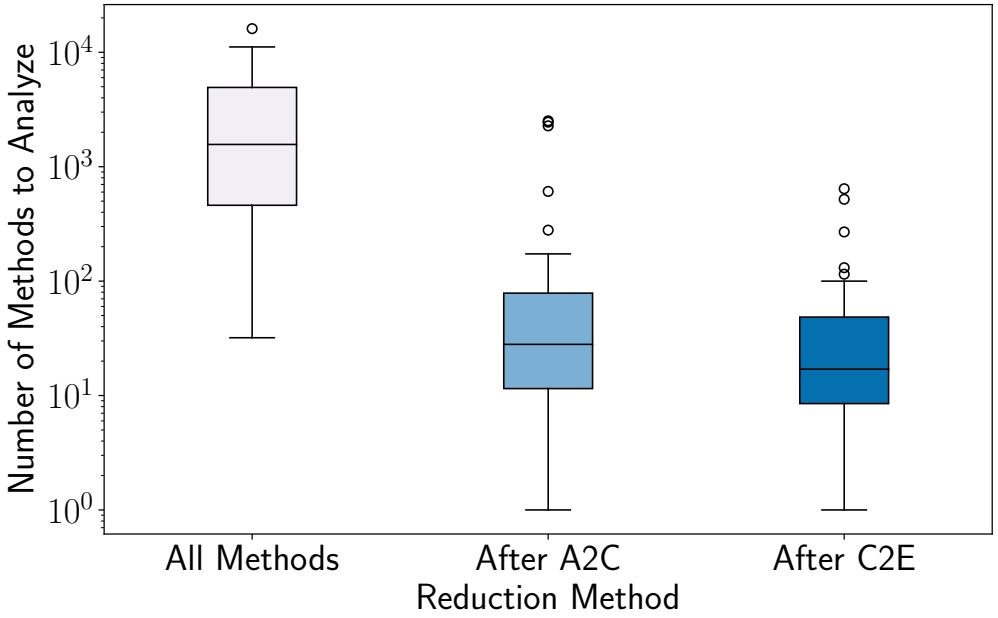


Figure 5.6.: Effect of the size reduction on the number of methods to be analyzed (ordinate is in log-scale) adapted from [NHSO21]

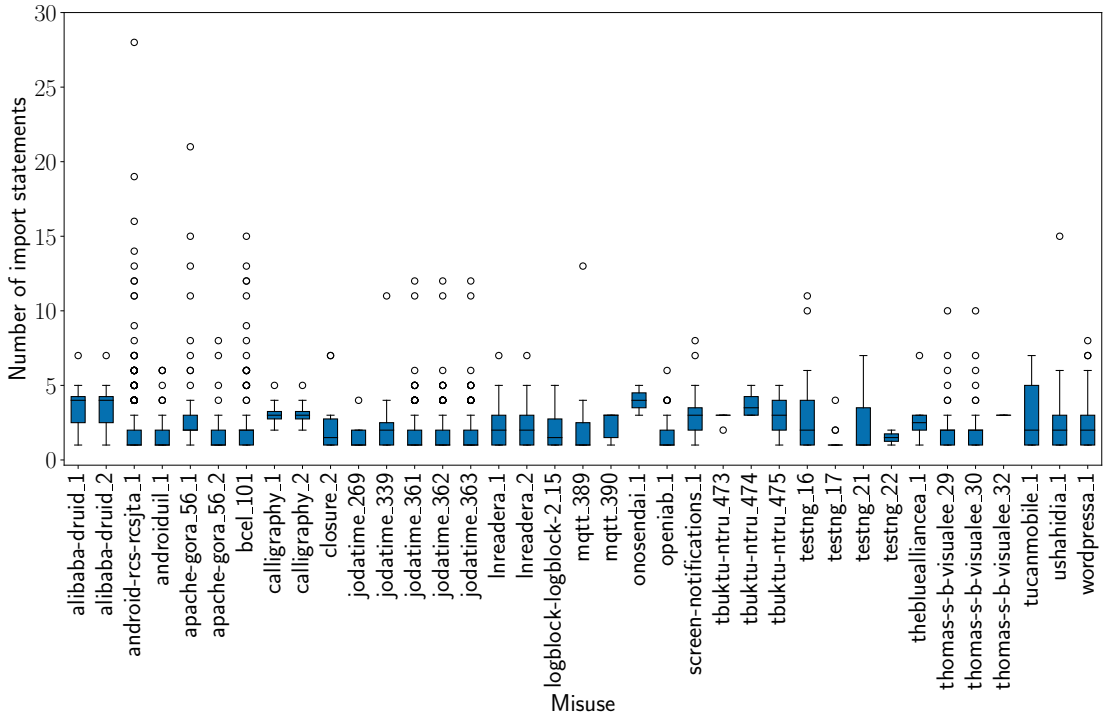


Figure 5.7.: Distribution of the number of import statements among the misuses for methods with at least one third-party import involved adapted from [NHSO21].

some huge outliers with up to 79 keywords remain.

Even though we did not precisely measure the execution time of the extraction and reduction, we estimated, based on the timestamps of the log files, that it lasted from a few seconds up to a few outlier cases with 3 minutes (e.g., `android-rcs-rcsjta_1`).

Implications Our results demonstrates that API change analysis effectively reduce the number of methods to be analyzed. Reduction by changed methods has a large effect on the remaining methods to be analyzed. Nevertheless, a qualitative view revealed that in cases where many methods were changed, we could further reduce the number by applying subsequent filtering if the method used a third-party API. Still, some extreme outliers (e.g., up to 642 methods) persisted and needed to be handled to avoid huge computational effort in subsequent pattern mining and misuse detection.

Insight D-6 (RQ D-C): Significant Reduction of Methods for API Misuse Location

We found that API change analysis using change information (i.e., commits and third-party APIs) effectively reduces the number of methods as potential misuse locations, most frequently from an order of magnitude of $2-4$ *to* $1-2$.

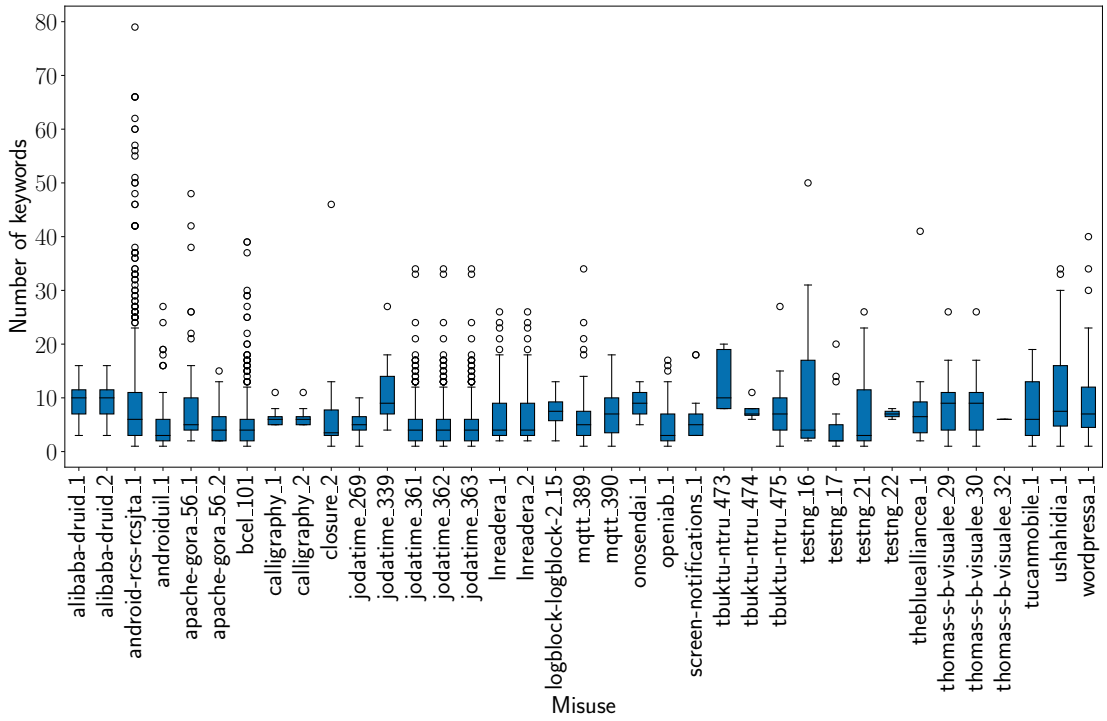


Figure 5.8.: Distribution of the number of extracted keywords among the misuses for methods with at least one third-party import involved adapted from [NHSO21]

For a majority of misuses (31 from 37), we correctly extracted at least one of the misused APIs. We further found that, on average, 1.8 third-party API import statements and 6.3 keywords were extracted per method. We hypothesize that these values, despite extreme outliers (e.g., `android-rcs-rcsjta_1` with 28 import statements and 79 keywords) per se do not indicate a negative effect on subsequent search and filter strategies, which will be evaluated in the next section.

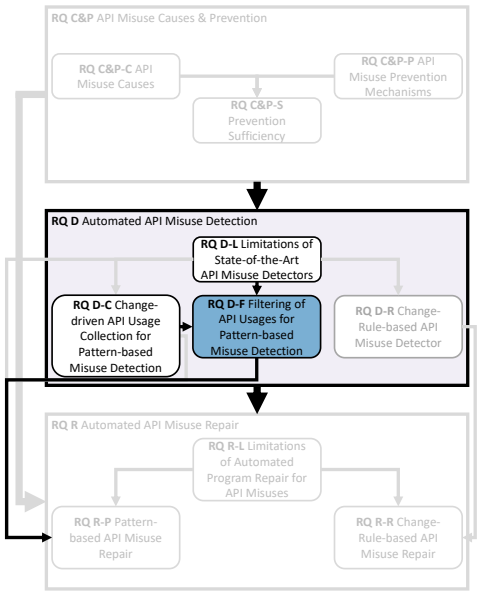
Insight D-7 (RQ D-C): Low Number of API Import Statements and Keyword for Client Code Search

Having, on average, 1.8 *third-party API import statements and* 6.3 *keywords denotes a low number, which we hypothesize to have no negative effect on a realistic code search scenario. Moreover, in 31 out of the 37 analyzed cases, we could successfully extract at least one misused API, increasing the chance of finding related API usages with subsequent searches.*

Table 5.8.: Different configurations for searching (i.e., $search_{loc}$ and $search_{imp}$) and filtering (i.e., $filter_{file}$ and method $filter_{method}$) adapted from [NHSO21]

Strategy	Configuration Options				
$search_{loc}$	internal	external			
$search_{imp}$	all imports	misused imports			
$filter_{file}$	sr=0.0	sr=0.25	sr=0.5	sr=0.75	sr=1.0
$filter_{method}$	applied	not applied			

5.5.2. Impact of Search and Filter Strategies on Mining Input



In this section, we present the validation of the impact of the different search and filter strategies on the input of API usage pattern mining. Our goal is to determine the best strategy, namely, this strategy that effectively discards that API donor code samples from an input set, which do not describe the correct usage, while keeping the related and correct ones. The best strategy increases the *relative frequency* of correct API usage patterns in the input set, and thus, support-based pattern mining approaches will likely rank correct patterns (i.e., those possible to detect misuses) higher. This way, we target the filtering effect on mining for **RQ D-F**.

Methodology We implemented the search and filter strategies (i.e., $search_{loc}$, $search_{imp}$, $filter_{file}$, and $filter_{method}$) in our replication package[24]. This way, we automatically validated different configurations by searching and filtering source files and methods for the 37 misuses from the MUBench dataset. In Table 5.8, we depict all configuration options we applied in our experiments. We varied the location to search (i.e., $search_{loc}$) between *internal*, namely, within the project containing the misuse, and *external*, namely in externally found source code according to the description in Section 5.3.2. As denoted before, for external search, we downloaded a consistent set of at most $2,000$ similar source files from *Searchcode* and applied all search and filter strategies locally. We used different sets of API imports when searching similar source files (i.e., $search_{imp}$). The first variant (i.e., *all imports*) uses all extracted import statements as search query. For the *misused imports* version, we were interested in whether a previous detection of the misused API improved the search results. This configuration emulated this behavior by conducting a separate search with the import statements of the misused API if this API was successfully extracted by the API change analysis. Note that we obtained the misused API(s) from the meta-data of the MUBench dataset. In the file filtering (i.e., $filter_{file}$), we varied the satisfaction ratio sr in the interval $[0, 1]$, while the extreme cases $sr = 0$ (i.e., no keywords have to be matched) and $sr = 1$ (i.e., all keywords have to be matched) served as baselines. We split this configuration into four equidistant

Table 5.9.: Configurations to obtain independent groups for statistical comparison adapted from our work [NHSO21].

Statistical comparison of	**Obtaining independent groups**
$search_{loc}$	no $filter_{file}$ (i.e., $sr = 0$); no $filter_{method}$; each single $search_{imp}$-strategy
$search_{imp}$	no $filter_{file}$ (i.e., $sr = 0$); no $filter_{method}$; each single $search_{loc}$-strategy
$filter_{file}$	no $filter_{method}$; each single $search_{imp}$ and $search_{loc}$ strategy
$filter_{method}$	no $filter_{file}$ (i.e., $sr = 0$); each single $search_{imp}$ and $search_{loc}$ strategy

sections, namely $[0, 0.25, 0.5, 0.75, 1]$, which allowed sufficient testing of different sr values while not heavily increasing the configuration space to test. In contrast, method filtering (i.e., $filter_{method}$) was rather simple by either *applying* it (i.e., method has to contain at least one keyword) or *not*. These variants formed a set of 40 configurations per misuse ($1,480$ for the 37 misuses from MUBench).

We measured the success of a single configuration by the *relative pattern frequency*, which denotes the proportion of how often a correct (i.e., misuse detecting) API usage pattern is present in the number of entries in the filtered input set. The relative pattern frequency was obtained as follows:

1. We manually distilled one or multiple correct API usages as ground truth usage based on the MUBench meta-data of the misuse.
2. We automatically transformed this ground truth usage into their corresponding AUGs.
3. We automatically transformed each method obtained from the search and filter configuration into their corresponding AUGs[25].
4. We automatically detected how often the ground truth AUG (or AUGs if multiple possible solutions exist) was a subgraph (i.e., subgraph isomorphism) in the set of the previously created AUGs.
5. We computed the *relative pattern frequency* as the largest value of the number of occurrences of a ground truth pattern divided by the number of AUGs present in the filtered set.

Note that the general subgraph isomorphism problem is known to be NP-complete [Epp99], and thus, we applied a simplified version by checking whether the nodes and edges of the ground truth AUG are a subset of respective AUG from the set of AUGs obtained through the single configuration. This heuristic effectively introduces an overestimation of the relative frequency since the ground truth AUG could be falsely detected as a subgraph. Thus, the real *relative pattern frequency* may be lower. We tested the significance of relative pattern frequency using the non-parametric Wilcoxon signed-rank test ($\alpha = 0.05$), which does not require a normal distribution [Kan06, p. 101]. This test, however, requires that the two paired groups are independent. This requirement did not

[25]To avoid frequent re-generation, we stored these AUGs permanently after first generation.

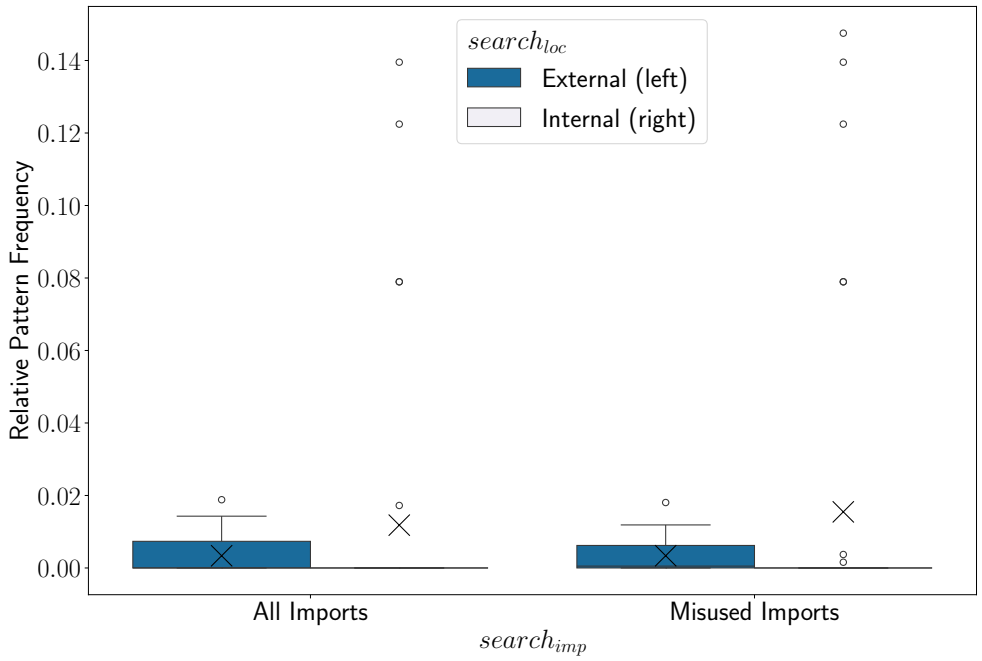


Figure 5.9.: Distribution of the relative pattern frequency using different file search strategies grouped by API search strategy adapted from [NHSO21].

hold for all configurations. For instance, when comparing the different configurations of the $filter_{method}$ strategy, of those 740 configurations applying method filtering, 592 of them used the same source files but with a different $filter_{file}$ configuration. Consequently, we ignored the other search and filter configurations when measuring the single effect of one strategy. The setting to obtain independent groups for the strategy comparisons is depicted in Table 5.9.

Results From all tested 1,480 configurations, we found 748, which retrieved at least one similar source file. From these similar files, 383 contained at least one occurrence of the ground truth AUG. Considering the 37 misuses tested, for 33 of them, at least one of the 40 configurations per misuse found at least one similar source file. For 22 misuses, at least one configuration obtained the ground truth API usage.

In the following, we explored the effect of the single strategies $search_{loc}$, $search_{imp}$, $filter_{file}$, and $filter_{method}$ first, by reporting the number of misuses for which the similar source files contained at least one the ground truth AUG, and second, by comparing and analyzing the *relative pattern frequency*.

Result $search_{loc}$***:*** First, we compared the *internal* and *external* $search_{loc}$. While the internal search found the ground truth AUG only for seven misuses, the external search found it for 22 misuses. We depict the distribution of the relative pattern frequency in Figure 5.9, comparing the two independent variants of the $search_{imp}$ strategy. In both cases, we observed that the mean relative frequency (marked as $\times$) of the internal search was larger than the one of the external search. However, the higher mean was caused by

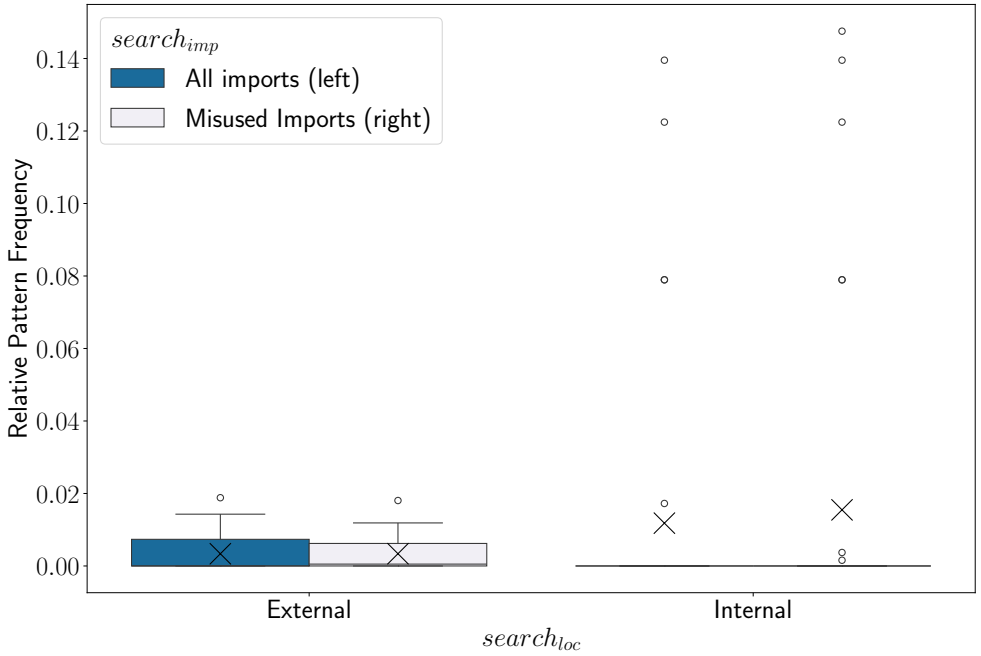


Figure 5.10.: Distribution of the relative pattern frequency using different API search strategies grouped by each file search strategy adapted from [NHSO21].

the outliers in the internal search. Consequently, we also found no significant difference in the distribution. Thus, we conclude that even though external search found more misuse, it is worth conducting an internal search as well, for instance, in a cascaded manner.

Result $search_{imp}$***:*** Second, we investigated the difference between using only *misused imports* and *all imports* $search_{imp}$. By the *misused import* search, we obtained the ground truth AUG for 22 misuses, while with *all imports*, we found the ground truth for 17 misuses. We compared the distribution of the relative pattern frequency among the $search_{loc}$ configurations in Figure 5.10 . In both single variants, the boxplots indicated almost identical distributions. This visual observation was also further supported by a non-significant Wilcoxon signed-rank test. Therefore, we conclude that conducting an analysis to find misused APIs has only a marginal effect, assuming one may find a technique to effectively identify the misused APIs.

Result $filter_{file}$***:*** In Table 5.10, we compared the effect of different *satisfaction ratios* (i.e., sr) on finding the ground truth patterns (i.e., $filter_{file}$). Unsurprisingly, the number of misuses, for which we obtained at least one ground truth AUG, is decreasing for increasing sr. Nevertheless, we observed a drastic drop between 0.75 and 1. Due to visibility reasons, we depict only the mean values of relative pattern frequency (cf. Figure 5.11)[26]. We observed that the relative frequency tended to be almost constant up to $sr = 0.5$, then slightly increased for $sr = 0.75$ and then dropped to its minimum for $sr = 1$. Moreover, the curves for the internal $search_{loc}$ also lie constantly above the external $search_{loc}$. However, we only found significant differences in the means of $sr = 1$ and all other sr-values (except

[26]The distributions can be found in the appendix (cf. Figure A.6 and Figure A.7).

sr	**Misuses with Ground Truth AUG found**
0.00	22
0.25	21
0.50	21
0.75	18
1.00	4

Table 5.10.: Number of misuses per satisfaction ratio *sr* for which at least one fixing pattern was found adapted from [NHSO21].

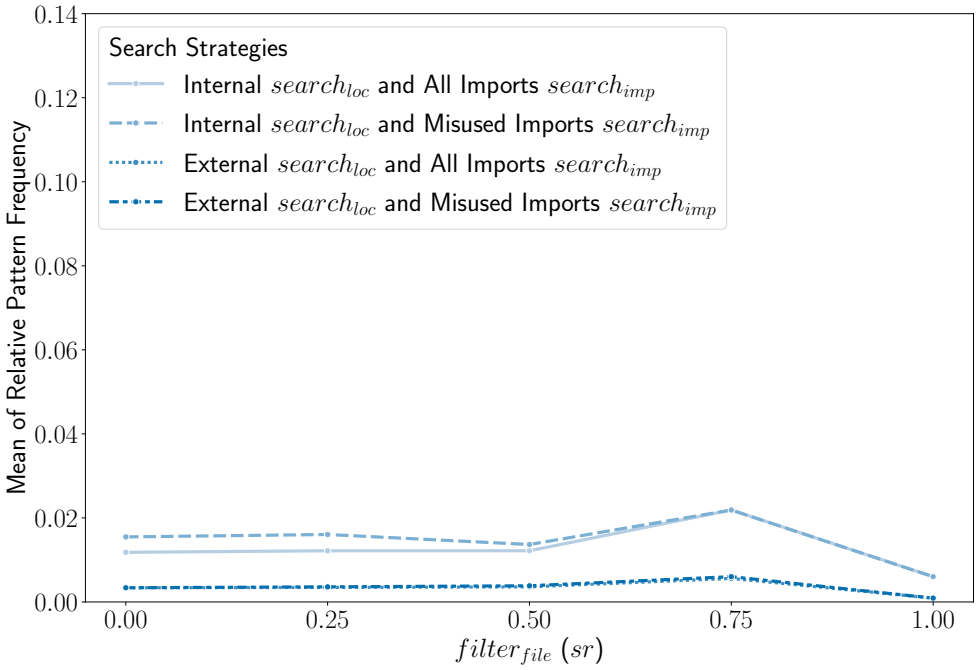


Figure 5.11.: Mean values of the relative pattern frequency among different file filter strategies (satisfaction ratio) grouped by certain search strategies adapted from [NHSO21].

for all *internal* $search_{loc}$ and *all imports* $search_{imp}$ configurations as well as for the comparison to the $sr = 0.75$ with *internal* $search_{loc}$ and *all imports* $search_{imp}$ configuration). This way, we can only conclude that using $sr = 1$ (i.e., matching all keywords) usually has a negative effect.

Result $filter_{method}$***:*** Finally, we assessed the effect of the $filter_{method}$ strategy. We found that applying method filtering still obtained the ground truth AUG for 21 misuses, while not applying did so for 22 misuses. Once again, we depict the distribution of the relative frequency among the different $search_{loc}$ (i.e., Figure 5.12a and Figure 5.12b) as well as among the different $search_{imp}$ strategies. Note that the ordinates of both sub-figures are different. All boxplots indicated a difference in the mean relative pattern frequency towards *applying method filtering*. We also found these differences statistically significant, while for the *internal* $search_{loc}$, the number of paired elements was too small and might violate the internal normal approximation of the Wilcoxon signed-rank test. Thus, those results should be taken with care. Nevertheless, overall, we attested the $filter_{method}$ strategy had a positive effect on the relative pattern frequency.

Our concept only requires a method to match at least one keyword for $filter_{method}$. While we did not test different sr values on the method scope, we measured the sr values of all single methods that contain at least one keyword (i.e., same sr definition as for $filter_{file}$ but only using the method declaration). We found that the average sr for internal and external $search_{loc}$ is ≈ 0.11 and ≈ 0.19. Assuming an average number of keywords of 6.3 (cf. **Insight D-7**), the average absolute number of matched keywords (i.e., the products of method scope sr and average number of keywords) ranges between ≈ 0.7 and 1.1. Thus, we conclude that matching one keyword is sufficient since otherwise, too many methods are discarded (i.e., more than average).

Overall, we found the best-performing configuration with a mean relative pattern frequency of ≈ 0.058 (i.e., among all misuses using internal $search_{loc}$, only misused import $search_{imp}$, applying $filter_{file}$ with $sr = 0.25$, and applying $filter_{method}$). For external $search_{loc}$ the best configuration used misused import $search_{imp}$, $filter_{file}$ with $sr = 0.075$, and applied $filter_{method}$.

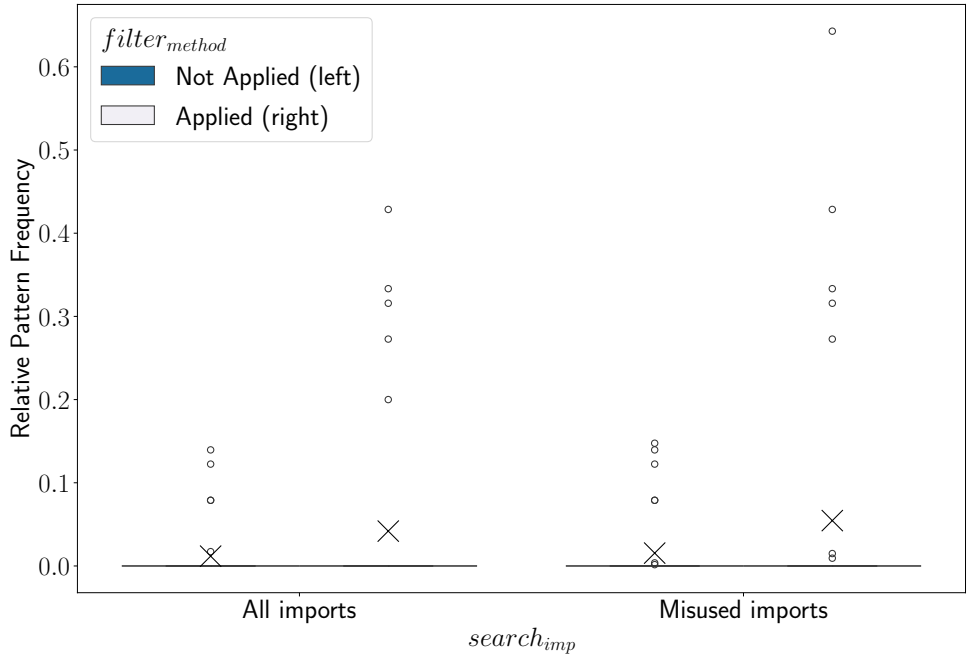
Finally, even though we did not specifically track the execution time, we could recall from the time stamps in the log files that search and filter strategies (excluding querying and downloading time from *Searchcode*) took at most two minutes per misuse.

Implications By validating the different configurations, we obtained insights into which search and filter strategies can produce the most promising results for usage pattern mining.

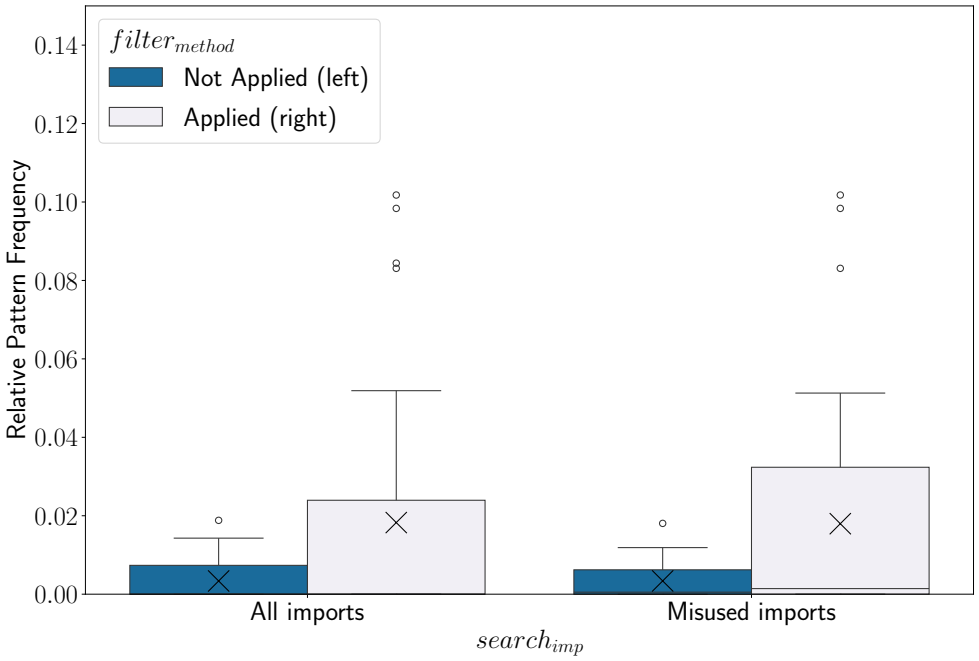
Particularly, we found that *internal* $search_{loc}$ did not find as many ground truth patterns as the external one. However, we could not determine a significant difference between both configurations. Since the internal code base is usually much smaller than the external one, we found it to be a valuable supplement and thus suggest a *cascaded usage* of the code search (i.e., first internal search followed by external search). This is possible since *internal* and *external* $search_{loc}$ were applied on independent sets.

Insight D-8 (RQ D-F): Usefulness of Internal Search for Finding Client Code

We observed that internal code search usually finds fewer code samples to obtain relevant API usage and patterns. Nevertheless, it provides a fast and useful supplement before conducting more extensive external code search.



(a) Internal File Search with an adapted ordinate for relative pattern frequency



(b) External File Search

Figure 5.12.: Distribution of the relative pattern frequency applying the method filter strategy grouped by each File and API search strategy both adapted from [NHSO21].

Next, we discovered that the effect of detecting and applying the misused API (i.e., $search_{imp}$ strategies) for code search was negligible. Thus, we support the usage of all extracted API imports. A possible reason why this still achieved acceptable results is that multiple imports can better describe the context of an API usage (e.g., which APIs are used together) while still having a moderate number of extracted `import` statements (i.e., on average 1.8 cf. **Insight D-7**).

Insight D-9 (RQ D-F): No Necessary Knowledge on the Misused API for Finding Relevant Client Code

We found that it was not necessary for our technique to know the misused API(s) to distill relevant client code and patterns for misuse detection. This result is most probably achieved by describing the context of the API usage by their correlated APIs and code keywords.

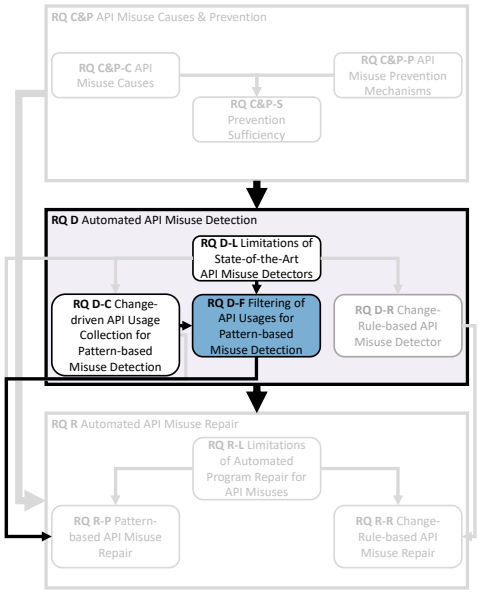
Regarding the filter strategies, we found only a moderate effect of the file filtering (i.e., $filter_{file}$). Particularly, sr-values up to 0.5 usually reduced the number of source files without discarding too many source files containing the ground truth pattern. Thus, we suggest applying the satisfaction ratio of $sr = 0.5$.

In contrast, the method filter (i.e., $filter_{method}$) has been proven far more effective with a positive impact on the relative pattern frequency. Thus, applying the method filter is promising for API usage pattern mining.

Insight D-10 (RQ D-F): Method Filtering over File Filtering

We found method filtering of client code using keywords from changed API usage was far more effective than filtering source files. This way, we increase the relative frequency of code snippets containing useful patterns.

5.5.3. Impact of Change-Based Inference on API Misuse Detection



In this section, we analyze the impact of the complete process chain of API change analysis with subsequent search and filter strategies and mining processes on API misuse detection. For this purpose, we analyze the misuse detection manually as well as with an existing automated API misuse detector, both with and without a filter strategy. We apply that search and filter strategy, which performed best in previous validation. This way, we investigate the effect of search and filter strategies on the overall misuse detection, and thus, we target **RQ D-F**.

Methodology As previously discussed (cf. Section 5.2), many misuse detection methods apply an FPM technique, namely, inferring interesting patterns as specifications of correct API usage and use these patterns to determine their violations as misuses. We already

discussed the numerous methods to mine patterns, both general ones (cf. Section 3.5.2) as well as for API usage patterns (cf. Section 3.5.5). In the evaluation of the misuse detection, we applied MUDetect by Amann et al. [Ama18, ANN+19b], who also applied an FGM technique. We configured this miner to apply the cross-method-based support (i.e., counting the number of distinct methods containing a pattern candidate), which also matched our definition of the *relative pattern frequency* used in the previous section.

In the first evaluation, we only applied the FGM from MUDetect based on the 22 misuses from MUBench [ANN+16], for which one of our tested configurations found at least one API usage representing the ground truth pattern. Particularly, we compared the mining results based on the *non-filtered* and the *filtered* dataset. For the filtered dataset (i.e., *our intervention*), we applied the configurations we suggested in the previous section, namely:

- $search_{loc}$: internal (I) and external (E) in a cascaded manner while we analyzed the effect of both configurations individually
- $search_{imp}$: we searched with all extracted API imports
- $filter_{file}$: we set the satisfaction ratio sr to 0.5
- $filter_{method}$: we applied the method filtering

Moreover, the mining algorithm requires a minimum support threshold (i.e., $min_{support}$), which we also derived from our previous analysis. In detail, we used the lower quartile of the distribution of the relative pattern frequency that is greater than zero from all configurations (i.e., filtered as well as non-filtered ones). This way, we obtained single values for internal (i.e., $min_{support} = 0.08$) and external $search_{loc}$ (i.e., $min_{support} = 0.004$). We distinguish between these two configurations since the number of internally obtained AUGs is much lower, and a too-small $min_{support}$ would eventually select all possible elements from the dataset as pattern candidate[27]. After mining, we sorted all pattern candidates according to their absolute support up to rank 20. In case multiple candidates had the same support, they share the same rank. Therefore, we enforced that the next subsequent candidate with lower support had the next rank increased by the number of candidates sharing the previous rank. Then, two assessors (i.e., the first two authors of the related paper [NHSO21]) manually and independently reviewed each pattern candidate and decided whether this candidate represents the ground truth pattern. Particularly, we distinguished between

- *is pattern* - the pattern inferred was equal to the ground truth and thus able to fix the misuse, and
- *is super-pattern or equivalent pattern* - the inferred pattern was either a super-pattern (i.e., ground truth AUG was a sub-graph) or the pattern described an equivalent pattern, which we did not assess as ground truth in the first place.

Note, in case we denoted the candidate as *is pattern*, it was automatically assessed as *is super-pattern or equivalent pattern*, and thus, the right-hand side of Table 5.12 and Table 5.13 depict the accumulated results. We chose the manual evaluation over the automated one since this also provided some qualitative insights, for instance, the necessity to analyze *super-patterns*. We measured the agreement of both assessors using Cohen's κ (cf.

[27] when configuring the miner in our implementation, we had to set the absolute minimal support, which we computed based on the relative value of the minimal support, and in case this value was lower or equal to one we set it to two

Table 5.11.: Agreement between the assessors when validating the pattern candidates for MUBench. The agreement for non-filtered, internal dataset is not available since we only obtained five negative results for which both assessors agreed.

dataset / **decision**	**non-filtered**		**filtered**	
	internal	*external*	*internal*	*external*
is pattern	NA	perfect ($\kappa = 1$)	perfect ($\kappa = 1$)	moderate ($\kappa = 0.43$)
is super-pattern or equivalent pattern	NA	almost perfect ($\kappa = 0.84$)	perfect ($\kappa = 1$)	substantial ($\kappa = 0.69$)

Table 5.11) and resolved conflicts in a discussion session, particularly for the moderate and substantial agreement.

In the second evaluation, we compared the filtered and non-filtered cases based on the AU500 dataset [KL21] by applying the exact same configuration for filtering as used before. Since this dataset was too large for manual inspection, we applied MUDetect directly. MUDetect mined patterns with an absolute minimal support threshold and ranked violations of these patterns by computing the overlap between API usage and pattern. For our validation, we applied the simple *overlap* function and its $cross - method$-variant as introduced in Section 5.2.4. Note since MUDetect accepted only absolute values for the minimal support thresholds, we set the thresholds to 2 for internal $search_{loc}$ and 10 for external $search_{loc}$ We had observed these as typical values in the previous validation. We also defined a timeout for mining of 5 minutes and 10 minutes using internal and external $search_{loc}$, respectively. Based on the ground truth labels in AU500 [KL21], whether the API usage represented a misuse or not, we defined the detection as *true positive* if the usage was labeled as *misuse* and the MUDetect finds a violation (i.e., the *overlap* function computes a value in the interval $[0, 1]$). Particularly, we checked whether at least one violation was found by MUDetect for the method declaration from AU500. If the usage was labeled as misuse and no violation was detected, we denoted this as a *false negative*. Similarly, a usage labeled as *correct* with a violation was denoted as a *false positive* and as a *true negative* if no violation was observed. Accordingly, we computed precision and recall beyond all results of the 500 API usages.

Results on MUBench We present the results of the manual pattern candidate analysis of the 22 misuses from MUBench for the *non-filtered* (cf. Table 5.12) and the *filtered* case (cf. Table 5.13). We distinguished between our decision of *is pattern* and *is super-pattern or equivalent pattern* and determined whether this pattern was ranked in the Top@k with $k \in \{1, 5, 10, 20\}$. In the non-filtered case (cf. Table 5.12), we first observed that only patterns with the *external* $search_{loc}$ were retrieved. In detail, the miner inferred the ground truth pattern (i.e., *is pattern*) for four misuses in the Top@10 and for eight misuses in the Top@20. While we found a *super-pattern or equivalent pattern* typically higher ranked (e.g., six and seven for Top@5 and Top@10, respectively), we also obtained these patterns for at most eight misuses in the *is pattern* case. In the filtered case (cf. Table 5.13), the miner inferred more ground truth or similar patterns than in the non-filtered case. Particularly, the *is pattern* case was found for seven (Top@10) and eight (Top@20) misuses. For *is super-pattern or equivalent pattern*, true patterns were obtained for ten (Top@10) and 13

Table 5.12.: Number of fixing patterns found in the Top@k patterns by mining ***without any filtering*** in the MUBench dataset from [NHSO21].

misuse		**pattern in Top**				**is super-pattern or equiv. pattern in Top**			
		@1	@5	@10	@20	@1	@5	@10	@20
$\sum = 22$		**3**	**3**	**4**	**8**	**3**	**6**	**7**	**8**
1	alibaba_druid_1	-	-	-	-	-	-	-	-
2	alibaba_druid_2	-	-	-	-	-	-	-	-
3	android_rcs_rcsjta_1	-	-	-	-	-	-	-	-
4	apache_gora_56_1	-	-	-	-	-	-	-	-
5	apache_gora_56_2	-	-	-	-	-	-	-	-
6	bcel_101	-	-	-	-	-	-	-	-
7	jodatime_269	-	-	E	E	-	-	E	E
8	jodatime_361	E	E	E	E	E	E	E	E
9	jodatime_362	E	E	E	E	E	E	E	E
10	jodatime_363	E	E	E	E	E	E	E	E
11	logblock_2_15	-	-	-	-	-	-	-	-
12	mqtt_389	-	-	-	-	-	-	-	-
13	mqtt_390	-	-	-	-	-	-	-	-
14	tbuktu_ntru_473	-	-	-	-	-	-	-	-
15	tbuktu_ntru_474	-	-	-	-	-	-	-	-
16	tbuktu_ntru_475	-	-	-	-	-	-	-	-
17	testng_16	-	-	-	-	-	-	-	-
18	thebluealliancea_1	-	-	-	-	-	-	-	-
19	thomas_s_b_visualee_29	-	-	-	E	-	E	E	E
20	thomas_s_b_visualee_30	-	-	-	E	-	E	E	E
21	thomas_s_b_visualee_32	-	-	-	E	-	E	E	E
22	ushahidia_1	-	-	-	E	-	-	-	E

E: pattern found in external source files;
I: pattern found in internal source files;
I/E: pattern found in internal and external source files

Table 5.13.: Number of fixing patterns found in the Top@k patterns by mining ***with our intervention***, namely, filtering of the MUBench dataset with the predefined configurations from [NHSO21].

misuse		**pattern in Top**				**is super-pattern or equiv. pattern in Top**			
		@1	@5	@10	@20	@1	@5	@10	@20
$\sum = 22$		**4**	**7**	**7**	**8**	**5**	**9**	**10**	**13**
1	alibaba_druid_1	-	-	-	-	-	-	-	-
2	alibaba_druid_2	-	-	-	-	E	E	E	E
3	android_rcs_rcsjta_1	-	-	-	-	-	-	E	E
4	apache_gora_56_1	-	-	-	-	-	-	-	-
5	apache_gora_56_2	-	-	-	-	-	-	-	-
6	bcel_101	-	-	-	-	-	-	-	-
7	jodatime_269	-	-	-	E	-	-	-	E
8	jodatime_361	E	E	E	E	E	E	E	E
9	jodatime_362	-	E	E	E	-	E	E	E
10	jodatime_363	E	E	E	E	E	E	E	E
11	logblock_2_15	-	-	-	-	-	-	-	-
12	mqtt_389	-	I	I	I/E	-	I	I	I/E
13	mqtt_390	-	I	I	I	-	I	I	I
14	tbuktu_ntru_473	-	-	-	-	-	-	-	E
15	tbuktu_ntru_474	-	-	-	-	-	-	-	E
16	tbuktu_ntru_475	-	-	-	-	-	-	-	-
17	testng_16	-	-	-	-	-	-	-	-
18	theblueallliancea_1	-	-	-	-	-	-	-	-
19	thomas_s_b_visualee_29	I	I	I	I	I/E	I/E	I/E	I/E
20	thomas_s_b_visualee_30	I	I	I	I	I/E	I/E	I/E	I/E
21	thomas_s_b_visualee_32	-	-	-	-	-	-	-	-
22	ushahidia_1	-	-	-	-	-	E	E	E

E: pattern found in external source files;
I: pattern found in internal source files;
I/E: pattern found in internal and external source files

(Top@20) misuses. Moreover, for both *internal* and *external* $search_{loc}$, we obtained results.

In general, we observed a positive effect of the filter strategy on the number of *super- or equivalent patterns*, while for the ground truth pattern the number of misuses, stayed the same. We quantitatively checked whether this difference was statistically significant by applying a χ^2-test ($\alpha = 0.05$) with Yates correction [Yat34] to handle multiple comparisons on the same data. This test revealed *no significant difference*.

Filtering seemed to enable the applicability of internally found pattern candidates, as we found patterns for four misuses when applying filtering compared to none in the non-filtered case. In three of those cases, both $search_{loc}$ strategies found true patterns in the Top@20 results. Thus, a previous internal search could avoid the necessity of a more expensive external search. In the cases of `thomas_s_b_visualee_29` and `thomas_s_b_visualee_30`, we observed that even though patterns from external $search_{loc}$ were not present in the Top@20 compared to the non-filtered case, with filtering this was compensated by a successful internal $search_{loc}$. This observation further supports the hypothesis to apply both strategies. Moreover, we conjecture for the non-filtered case that it profits from searching with API `import` statements, which to some degree also filters the input data.

On the other hand, we observed that filtering could also discard true positive patterns when applying the external $search_{loc}$ strategy. This behavior occurred for `ushahidua_1` for which we did not observe any *is pattern* results, for `thomas_s_b_visualee_32` where filtering found no patterns at all, and for `jodetime_269`, the ranks of true positive patterns decreased.

Additionally, we analyzed the reason why filtering could not infer the pattern for nine misuses. For three of them (i.e., `apache_gora_56_2`, `testng_16`, and `thomas_s_b_visualee_32`), the miner inferred no pattern at all (i.e., no pattern achieves achieves the minimum support threshold). This effect of no inference was stronger for internal $search_{loc}$ (for 17 misuses) than for the external one (only those three mentioned misuses). For the other six misuses, in at least one of the $search_{loc}$ variants, the filtering reduced the number of occurrences of the true positive patterns or removed all of them so that the support value decreased and the true positive pattern was not present in the Top@20 pattern candidates.

During analysis, we noticed that many pattern candidates *almost* matched the ground truth pattern but missed some essential parts. This observation indicated that the miner tends to over-simplify the candidates. Additionally, we also observed many very similar pattern candidates among the Top@20 results. By applying a clustering technique, such as the one by MAPO [ZXZ+09], the patterns and their support values could be summarized, increasing the chance of finding more distinct and frequent patterns.

The mining without filter tool took around one minute per misuse (estimated via the log files), while we found some extremes with 18 minutes (e.g., `thomas_s_b_visualee_32`).

Results on AU500 In the second evaluation, we compared the non-filtered and the filtered dataset by applying MUDetect [Ama18, ANN+19b] on the AU500 dataset [KL21]. The results are summarized in Table 5.14, where we depict the values for internal and external $search_{loc}$ individually as well as together (i.e., *both*). Note that this row does not necessarily represent the sum of the upper ones but the union, meaning if with both strategies (i.e., internal and external), the misuse is correctly detected, it is counted only once. For technical reasons, we could not apply MUDetect for 20 API usages from AU500, and consequently, we counted them as negative results (i.e., no violation was detected). That means depending on their ground truth label (i.e., misuse or correct usage), they

Table 5.14.: Results of the Misuse Detection on the AU500 dataset using the violation detection technique from MUDetect with number of true positives (#tp), false positives (#fp), true negatives (#tn), false negatives (#fn), precision, and recall adapted from [NHSO21].

Applied filtering	$search_{loc}$	**#tp**	**#fp**	**#tn**	**#fn**	**precision**	**recall**
No	external	8	16	369	107	33.33%	6.96%
	internal	1	10	375	114	9.09%	0.87%
	both	9	26	359	106	25.71%	7.83%
Yes (**Our intervention**)	external	8	12	373	107	40.0%	6.96%
	internal	9	27	358	106	25.0%	7.83%
	both	13	31	354	102	29.55%	11.3%

were counted as *true negative* (i.e., #tn) or *false negative* (i.e., #fn).

Overall, we observed a positive effect of filtering for both precision (+3.84%) and recall (+3.47%). Particularly, the results for internal $search_{loc}$ improved over the non-filtered case. However, using a χ^2-test, we found the differences not statistically significant.

Additionally, we found that the improvement tended to be an effect of reducing the number of input AUGs for mining. For instance, for internal $search_{loc}$ in the non-filtered cases, only for 20 API usages the mining run actually finished, while for the filtered case, this number increased to 453. Second, when considering the external $search_{loc}$ case, we only observed an improvement in the precision by slightly reducing the number of false positives while the recall stays the same. An interesting insight was that even though the number of true patterns was equal for the filtered and non-filtered cases, the actual detected misuses differ. Particularly, both found misuses for three API usages that the respective other case did not detect. For the non-filtering case, this was caused by a timeout during mining, while for the filtering one, too many true positive occurrences were removed, and the true positive pattern support fell below the minimal support threshold.

Implications In both evaluations, on MUBench [ANN+16] and AU500 [KL21], we could not determine a significant statistical effect on misuse detection. However, we found that the filtering strategy tends to have a positive effect on detecting misuses with internally inferred patterns. We observed that due to the reduction of the input data through filtering, the chance of finishing mining runs increased while simultaneously the risk of discarding too many true positive patterns increased as well. Thus, we observed only a minimal overall effect.

Insight D-11 (RQ D-F): Small Positive Effect of Searching and Filtering for Pattern-based API Misuse Detection

Even though we did not find a statistically significant difference, we observed for both validated benchmarks (i.e., MUBench and AU500) an increase in precision and recall of the MUDetect [ANN+19b] misuse detector when applying search and filter strategies based on change information. Moreover, from a qualitative analysis of the results for the MUBench dataset, we conjectured a positive effect on the pattern quality.

We compared these results also in the light of the analysis by Kang et al. [KL21], who analyzed MUDetect (or, more precisely, a variant named MUDetectXP using external code

for mining) and their active learning-based approach ALP also on the AU500 dataset. In their experiments, they applied an external code search, meaning retrieving API usage examples from external projects. In a first variant, they reused a dataset by Amann et al. [ANN+19b], which applied the BOA [DNRN13] to find 1,000 API usages from GitHub by explicitly searching API elements of the misused API. Kang et al. obtained by applying MUDetect on AU500 a precision of 27.6% and a recall of 29.6%, while ALP achieved a precision of 28.2% and a recall of 58.3%. Thus, we obtained a slightly larger precision (i.e., 29.55%) but a lower recall (i.e., 11.3%) using both $search_{loc}$ strategies (cf. Table 5.14). They also obtained their own dataset of API usages by applying AUSearch [ATLJ20], a tool constructed to find API usage examples by querying concrete API elements (e.g., method and class name). For each used API in the AU500 dataset, they obtained 2,330 examples on average. Using ALP, they obtained a precision of 44.7% and a recall of 54.8%.

In contrast to our approach, their external search leveraged the fact of knowing the used API and specifically searching API usage examples for that purpose. While our results of the obtained precision, particularly for the external $search_{loc}$ (cf. Table 5.14), were comparable to those achieved by MUDetect and ALP, the recall was drastically smaller. Thus, in practice where we hardly know the misused API, misuse detection could drastically lose precision. Our experiments on search and filter strategies revealed that precision and recall could be slightly (cf. **Insight D-11**) increased even though not statistically significant.

5.5.4. Threats to Validity

Due to the empirical nature of our validation, we discuss potential threats to *internal* and *external* validity, as presented in Section 5.1.

Internal Validity The first internal threat denotes the similar code found via external resources (i.e., external $search_{loc}$) since it may not be present at the time the misuse has been committed (i.e., temporal bias). Thus, in practice, no true positive pattern would have been found. This situation may happen if an already fixed version of the misuse was cloned by another project, either by copying or forking the project. While we diminished this effect by removing source files having the same `package` statement, still source code that was generated after the misuse introducing commit may be present in our data.

Moreover, the results may be biased by the performance of the underlying search engine *Searchcode*. We handled this issue by downloading the externally found source code and thus applying subsequent search and filter strategies under the same initial conditions for all configurations using external $search_{loc}$.

We can neither guarantee that patterns are correct nor that they are complete. Ground truth patterns may not be correct since we cannot guarantee that these patterns fix the misuse completely or whether they introduce new errors through confounding effects. In our qualitative validation of the misuse detection for MUBench (cf. Section 5.5.3), we have seen that the ground truth patterns were not complete since we found a set of *equivalent patterns*. Thus, it may be that we still missed some further true positive patterns, which would otherwise increase the relative pattern frequency.

Using the *relative pattern frequency* may be too skewed for the low number of AUGs. That means with a low number of AUGs, it is more likely to achieve a larger relative pattern frequency than with a large number of AUGs. We have seen this effect in the difference of

the internal and external $search_{loc}$. Thus, the observed differences may be only present by chance.

Finally, we conducted a subjective assessment of the pattern candidates for the MUBench dataset. Even though both assessors independently of each other validated the candidates and we measured the agreement, both researchers still belong to the same research group with a similar research focus. Thus, some subjectivity, which naturally occurs in qualitative studies, may be present. For that purpose, we published all our results and data in our replication package[24].

External Validity Our API change analysis and subsequent API usage pattern mining and misuse detection only apply to intraprocedural API misuses and only to those misuses that can be found by static code analysis. Thus, we cannot directly relate our results to misuses, which are scattered among multiple method declarations. Our technique does not directly target dynamic misbehavior such as misuses causing Heisenbugs [RHR+25] as well as misuses caused by misconfiguration of external libraries through external files.

We adapted the MUBench dataset due to potential biases (cf. Section 5.4.1). Nevertheless, we can hardly conclude to which degree the results based on MUBench can be representative of other API misuses. This issue was also noticed by Sven Amann himself, denoting that "*[t]he benchmark dataset may not be representative for API misuses in the wild[...]*"[Ama18, p. 75]. We coped with this issue by applying the search and filter strategies on the AU500 dataset by [KL21], however, without applying the same qualitative analysis. Thus, these insights may not be directly applicable to other datasets. Moreover, our results may be reproduced or disproved on other datasets, for instance, with our own AndroidCompass dataset [NBKO21b] or a dataset of APIARTy, a program repair framework [KMSH21]. Note that the APIARTy dataset also contains misuses from MUBench.

Finally, we restricted our technique to the object-oriented programming language Java. In detail, the intermediate source code representation, as well as the API change analysis, leverage certain features of Java. Thus, it requires similar concepts and features in other programming languages and paradigms to generalize our results for them.

5.6. Summary Pattern-Based API Misuse Detection

Summary First, we analyzed the limitations of state-of-the-art API misuse detectors based on an SLR process targeting **RQ D-L**. In detail, we analyzed the literature and conducted a forward snowballing based on two related studies on automated API misuse detectors by Robillard et al. [RBK+13] and Amann et al. [ANN+19a]. Based on the set of publications, we inferred a list of misuse detectors and experiments analyzing their obtained precision and recall. We further qualitatively analyzed the strengths and weaknesses of these detectors to derive their limitations.

Second, we targeted the limitations of pattern-based API misuse detectors by our technique, named *RAIX*, targeting **RQ D-C** and experiments on different *search and filter strategies* with a focus on **RQ D-F**. In detail, RAIX analyzes API-specific changes and extracts API-specific information. We confirmed that commits were a valuable source for RAIX and that RAIX effectively reduced the number of potential misuse locations. We further validated that the API-specific information supported our search and filter strategies to find similar API usage examples. We determined the best configuration of the search and filter strategies for finding relevant API usages and obtaining the best results on subsequent API misuse detection. In detail, our strategies increased the quality of API usage patterns obtained by the API specification mining from MUDetect [ANN+19b] and increased precision (i.e., a large proportion of correctly identified misuses in the set of identified misuses) and recall (i.e., a large proportion of detected misuses from a set of known misuses) of the pattern-based API misuse detector MUDetect.

Contributions RQ D-L Regarding the limitations of state-of-the-art misuse detectors, we found that while many misuse detectors exist and reuse existing benchmarks, the benchmarks differ due to variances in the datasets or the requirements of the misuse detectors (***Insight D-1*** *'Non-uniform Benchmarks and Comparison' on page 97*). Thus, much research on misuse detection cannot be directly compared to the results of other publications. To compare to state-of-the-art, we stated that our own experiments were necessary, which require reusable replication packages. However, we only found a replication package for 43.7% of the analyzed detectors (***Insight D-2*** *'Non-available Replication Packages' on page 98*). This result implied that many detectors could not be directly reused, which prohibited fair comparison. Moreover, we observed that many misuse detectors applied API usage client code (i.e., 90% of the analyzed detectors) as donor code to infer API usage specifications. However, their processes to collect client code were based on impractical assumptions (e.g., searching with the misused API, which is not known at the time of the misuse), as discussed in ***Insight D-3*** *'Limited Practical Applicability to Collect Client Code' on page 102*. In addition, we found fewer misuse detectors ($\approx$ 6.6% of the analyzed detectors) leveraging additional information and, if so, only in a restricted manner (e.g., only using exception handling code) as denoted by ***Insight D-4*** *'Restricted Usage of Additional Information for Finding External Client Code' on page 103*). Finally, we discussed promising misuse detectors based on their reported results, however, we excluded many for comparison due to their targeted programming language, less promising results, non-available replication packages, and their effort to infer API usage specifications (cf. ***Insight D-5*** *'Limited Applicability of API Misuse Detectors for Comparison' on page 107*).

Thus, we answer **RQ D-L** as follows:

RQ D-L What are the limitations of state-of-the-art pattern-based API misuse detectors in practical scenarios?

We found that many state-of-the-art misuse detectors (1) have been evaluated with non-uniform benchmarks, and (2) their results are not replicable due to missing replication packages. They further (3) had limitations in collecting client code as a basis of specification inference in a realistic manner and (4) seldom leveraged further information apart from the misused API. Finally, (5) some obtained low precision and recall as well as required a large effort to infer specifications for detection.

Contributions RQ D-C A possible way to overcome these limitations is change-based information from commits. By extracting API- and change-specific information using the VCS with our technique RAIX, we enable the embedding of this process into a realistic CI development process. We analyzed the typical size of misuse-introducing commits and found a significant reduction of code locations for misuse detection analysis (cf. ***Insight D-6** 'Significant Reduction of Methods for API Misuse Location' on page 122*) with a reduction of method declarations from an order of magnitude of 2-4 to 1-2. Additionally, we observed that extracted search terms (i.e., API-specific keywords and import statements) obtained a small number of search terms, which strengthens their applicability for subsequent code search (***Insight D-7** 'Low Number of API Import Statements and Keyword for Client Code Search' on page 123*).

Thus, we answer **RQ D-C**:

RQ D-C Is change information a meaningful source for finding related API usage samples for API misuse detection in practical use cases?

API- and change-specific information are valuable and effective means to restrict the number of potential API misuse locations in a client code. Since this search process can be embedded into a CI process, it has a high potential to seamlessly integrate into a practical development scenario.

Contributions RQ D-F Finally, we analyzed whether and which search and filter strategies obtained the best result to support pattern-based misuse detection. For search strategies, we found that using the same project containing the misuse (i.e., *internal* $search_{loc}$) as well as other projects (i.e., *external* $search_{loc}$) contributed to finding true positive patterns with API specification mining (cf. ***Insight D-8** 'Usefulness of Internal Search for Finding Client Code' on page 129*). This insight partially contradicts the observations by Amann et al. [Ama18, ANN+19b], who found non-promising results for internal search. We observed that, particularly, the internal $search_{loc}$ benefited from the search and filter strategies. Thus, we conclude that these strategies make internal code search for API specification mining applicable.

Additionally, we investigated the effect of knowing the misused API on pattern mining by emulating a perfect misused API predictor. While we observed better results than using all changed APIs, we found that the difference is negligible (***Insight D-9** 'No Necessary Knowledge on the Misused API for Finding Relevant Client Code' on page 131*). We conjectured that using multiple changed APIs better describes the context of API usages and thus

compensated for the missing knowledge of the exact misused API. Assuming identifying the misused API is effortful and complex, this step can be left out.

The filtering step using API-related keywords from the API change analysis yielded less promising results for filtering on file than on method scope. This result may be caused by the more fine-grained level on which filtering is conducted and that the specification mining analyzed API usages on an intra-procedural level (cf. ***Insight D-10*** *'Method Filtering over File Filtering' on page 131*).

According to the overall misuse detection applying different settings, we observed a slight improvement using search and filter strategies in the precision and recall, which, however, was not statistically significant (***Insight D-11*** *'Small Positive Effect of Searching and Filtering for Pattern-based API Misuse Detection' on page 137*).

Thus, we conclude for **RQ D-F**:

RQ D-F What is the impact of the previous filtering of the donor code from which API patterns are mined on the subsequent pattern-based API misuse detection?

We found that API usage pattern inference for API misuse detection could be improved by (1) using internal (i.e., from client code) and *external (i.e., from other projects) donor code, (2) not necessarily requiring the knowledge of the misused API, and (3) applying filtering on method than on file-level. This yielded (4) a small positive effect on pattern-based misuse detection.*

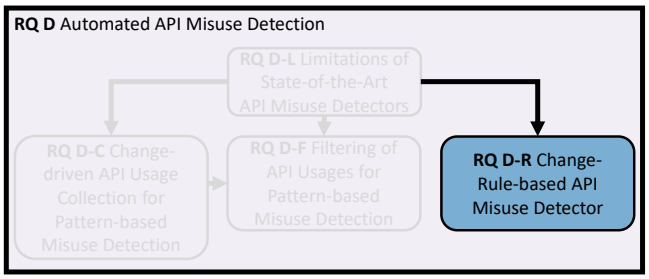
6

Change Rule-Based API Misuse Detection

This chapter is based on publications from the author together with other colleagues presented at the International Conference on Software Engineering (ICSE) 2020 in the New Ideas and Emerging Results Track [NHKO20a], presented at the International Working Conference on Source Code Analysis and Manipulation (SCAM) 2021 in the Replication and Negative Results Track [NBKO21a], and a non-peer-reviewed pre-print [NBKO22] discussed in the poster session of the Summer School on Security Testing and Verification 2022[1].

In this chapter, we contribute our own Application Programming Interface (API) misuse detector based on change rules from previous API misuse fixes as a further improvement to pattern-based misuse detectors discussed in Chapter 5. Both kinds of misuse detection become necessary if the root causes discussed in Chapter 4 cannot be prevented. The change rules introduced in this chapter are applied for the automated repair of API misuses in Chapter 7.

6.1. Methodology and Structure



This chapter targets the issue of low precision of many API misuse detectors [LHX+16, ANN+19a] and thus one part of **RQ D**. Since we observed limited applicability and replicability of misuse detectors, which reported a large precision (i.e., **Insight D-2** and **Insight D-5**), this issue has still a large relevance. Thus, we suggest a *change rule-based API misuse detection* and specifically target the sub-research question **RQ D-R**.

Again, we apply the scientific methodologies for the field of software engineering by Ralph et al. [RAB+20] and answer **RQ D-R** by an engineering research, namely conceptualize, implement, and evaluate a software artifact of inferring the change rules and applying them for a change rule-based misuse detection.

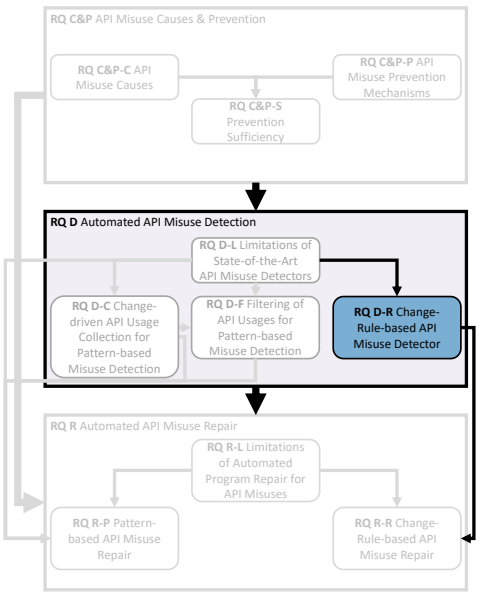
For this purpose, we first recall issues of the low precision of current API misuse detectors in Section 6.2. Then, we introduce the concept of change rules, their inference with an artifact named *Change Rule Inference (ChaRLI)*, and how to conduct a change rule-based

[1] https://cybersecurity-research.be/summer-school-program last accessed: 2025/02/05

API misuse detection with a technique named *Change Rule-based API Misuse Detection (RuDetect)* in Section 6.3. In Section 6.4, we present the experimental data and setting for the evaluation of ChaRLI and RuDetect, whose results are presented in the subsequent Section 6.5. In Section 6.6, we give a conceptual comparison to related misuse detectors, especially to those to which we could not compare in an experimental manner, as well as to API migration techniques (cf. Section 3.4.3). We summarize the results and the impact regarding **RQ D-R** in Section 6.7.

6.2. Imprecise API Misuse Detection Due to Alternative Usage and Missing Context

One recurring problem for state-of-the-art pattern-based misuse detectors is that *too many possible correct usages* exist, and thus, choosing a pattern that does not match the present context of an actual correct API usage results in a false positive. Therefore, collecting a set of patterns and applying the most frequent one is not sufficient for precise API misuse detection.



This issue was also observed by Amann et al. [ANN+19a], who found that uncommon but correct usages resulted in 34.3% and alternative pattern usages resulted in 19.2% of false positives in their analysis of misuse detectors on MUBench. Thus, if a certain usage is different from a pattern this does not necessarily give evidence of whether this usage is worth changing since this difference may still be correct.

In contrast, we conjecture that an API usage, which was found to be changed, gives far more evidence that similar usages at hand are also worth changing. These changes to API usages can represent API updates or migration steps as well as fixes of misused APIs.

Moreover, Amann et al. [ANN+19a] also found that many misuse detectors *miss important contexts*, for instance, corresponding objects and parameters of method calls. For that purpose, they applied the API Usage Graphs (AUGs) to more precisely describe the API usage and improve misuse detection [ANN+19b]. Thus, we will apply AUGs as our base data structure.

6.3. RuDetect: Concept of Change Rules for API Misuse Detection

In this section, we introduce RuDetect with its integrated technique ChaRLI to tackle the previously discussed limitations of API misuse detectors.

6.3.1. Overall Process

We conceptualize the notion of leveraging past changes to API usages to detect other similar API misuses as our technique named *Change Rule-based API Misuse Detection (RuDetect)*.

For this purpose, we introduce *API change rules*. These rules stem from past changes made by human developers, and thus, we conjecture that they denote valid changes that are able to decrease the number of false positives during misuse detection. We describe the aspired process of change rule inference and API misuse detection in Figure 6.1.

In practice, we assume a use case in which another developer has already fixed an API misuse in their code and committed this change through a Version Control System (VCS) and share their fix with others. This is frequently leveraged by historical-based automated repair techniques [KPW06, LLLG16, HW18]. In case they share their fix, we require a minimal manual effort of marking the method that has been fixed, the repository URI, the fixing commit hash, as well as the file path to the misuse (i.e., step (A)).

Afterward, the change rule is automatically obtained using our automated technique named *Change Rule Inference (ChaRLI)*. First, ChaRLI conducts a commit analysis in step (B) by extracting the source code of the misused file of the fix and the previous misuse revision using the VCS and generating a configuration for the subsequent AUG generation applying the technique by Amann et al. [ANN+19b]. This generation (i.e., step (C)) produces the AUGs for the misused and fixed code version. Finally, in step (D), we use these two AUG versions to create the change rule. The details of this generation will be described in the next Section 6.3.2. So far, note that this change rule represents the essential changes made to the API misuse (i.e., AUG aug_m) to transform it into a fixed version (i.e., AUG aug_f). Particularly, it removes non-necessary aspects of the change, namely, it creates a rule $aug_{m'} \rightarrow aug_{f'}$ where $aug_{m'}$ and $aug_{f'}$ denote subgraphs of aug_m and aug_f, respectively. This way, we make the change rule more likely applicable for misuse detection for other API usages.

The generated change rules are subsequently used to detect API misuses in other API usages. For that purpose, we first check whether the respective change rule is *applicable* for misuse detection ((E)). We present and validate two variants of the applicability check (cf. Section 6.3.3). In case the rule is not applicable, no decision will be made.

In case the rule is applicable, we use a graph similarity-based API misuse detection (cf. Section 6.3.4), which computes additionally the similarity *sim* between the usage graph aug_u and the two subgraphs $aug_{m'}$ and $aug_{f'}$ from the change rule in step (F). For the misuse detection step (i.e., (G)), we compare those two similarity values to determine whether the usage u is more similar to the misuse part m' than to the fix part f' of the rule (i.e., the similarity to $aug_{m'}$ is larger than to $aug_{f'}$). In this case, we denote the usage u to be a misuse. Otherwise, the usage u seems to be already fixed and, thus, is marked as correct.

Note that in case of multiple applicable rules, the similarity can be leveraged as a selection or ranking mechanism by picking that change rule for misuse detection having the largest similarity between aug_u and $aug_{m'}$.

6.3.2. ChaRLI: Change Rule Inference

Change Rule The change rule is based on AUGs, a data structure introduced by Amann et al [ANN+19b], which we already presented in Section 3.2.2. Particularly, ChaRLI leverages the code changes applied to an API usage as depicted in our sample code in Code Listing 6.1 and generates two sub-AUGs, one representing the pre-change version (i.e., the misuse) and the other the post-change version (i.e., the fix) whose nodes are connected by special `transform`-edges. Two variants (i.e., *with* and *without context* discussed subsequently) of generated change rules are represented in Figure 6.2 for the sample code in Code Listing 6.1.

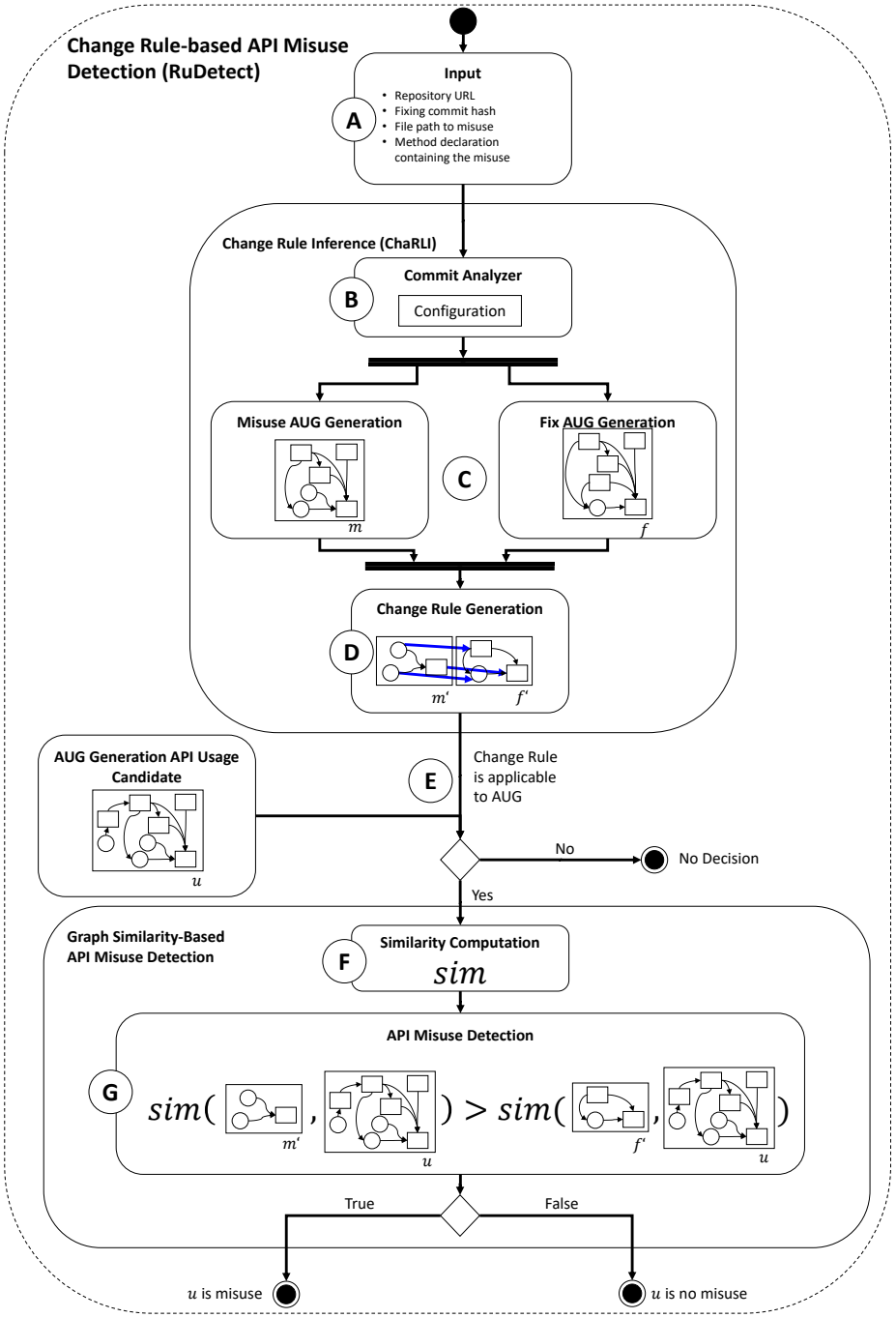
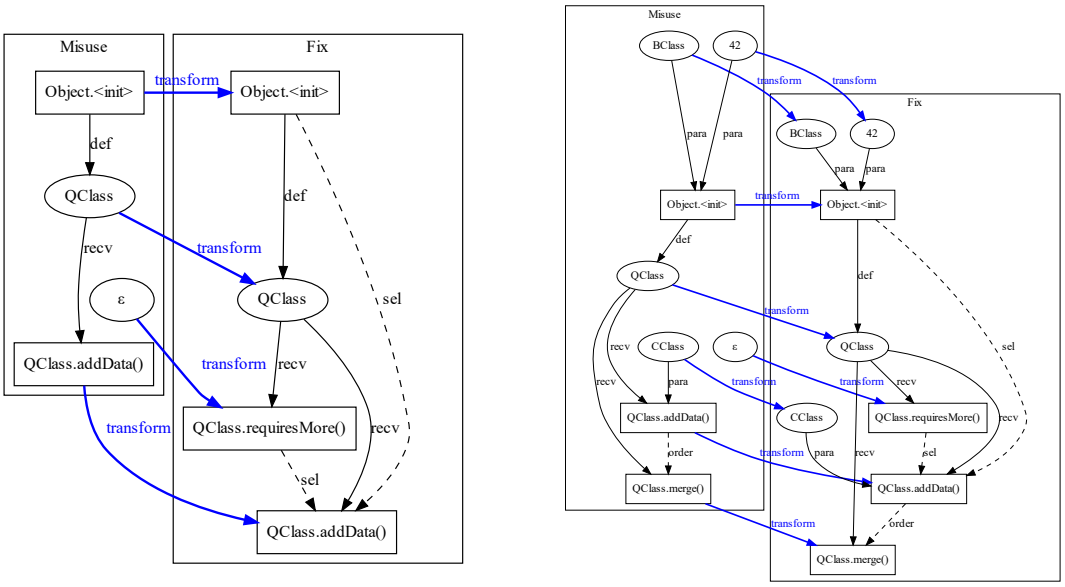


Figure 6.1.: Process of RuDetect by applying ChaRLI with subsequent Graph Similarity-based API Misuse Detection

```
1  package my.own.pkg.subpkg;
2
3  import a.b.AClass;
4  import a.b.BClass;
5  import a.b.CClass;
6  import x.y.ZClass;
7  import my.own.pkg.QClass;
8
9  public class Foo extends AClass {
10
11  BClass myB = new BClass(1337,"Bar");
12  CClass myC = new CClass();
13
14  protected ZClass doSomething(BClass bObj) {
15   System.out.println("do Something")
16   QClass myQObj = new QClass(myB, 42);
17 -  myQObj.addData(myC);
18 +  if(myQObj.requiresMore()){
19 +   myQObj.addData(myC);
20 +  }
21   return myQObj.merge(bObj);
22  }
23 }
```

Code Listing 6.1: Source Code Changes applied to an API Usage.



(a) Without Context. (b) With Context.

Figure 6.2.: Different Versions of API Change Rules for the Code Change in Code Listing 6.1.

These rules describe how nodes in an AUG have to be changed (i.e., relabeled and reconnected with the respective edges) to obtain the fixed version. In case a node is added or removed (e.g., adding or removing an API method call), we denote this by special ϵ-nodes representing 'holes' in the respective sub-AUG. For instance, in Figure 6.2, the action node `QClass.requiresMore()` is added in the respective fix AUG, and thus, there exists an ϵ-node in the respective misuse AUG. Analogously, if a node is deleted, the ϵ-node is present in the fix AUG.

Problem of Change Rule Inference In Figure 6.2, we depict two variants of change rules based on the changed code snippet in Code Listing 6.1, whose characteristics will be described successively. Regardless of their differences, we observe that both variants remove certain non-changed code, such as the `System.out.println`-method call in line 15 of Code Listing 6.1. More generally, we denote the change rule inference as function *charli* whose input are two subsequent AUGs aug_m and aug_f representing the misuse and its corresponding fix, respectively:

$$charli(aug_m, aug_f) = aug_{m'} \rightarrow aug_{f'}, \text{ where } aug_{m'} \subseteq aug_m \wedge aug_{f'} \subseteq aug_f$$

The problem of inferring change rules is similar to finding a *minimal mapping* between two AUGs or computing the minimal *edit path* of the minimal Graph Edit Distance (GED). Informally, computing the minimal GED between two graphs, g_A and g_B, computes the minimal costs of adding, deleting, or updating (i.e., relabeling) nodes and edges in a graph g_A to transform it into graph g_B [SF83]. The sequence of edit operations to obtain this minimal GED is then denoted as a minimal *edit path*. The one-to-one mapping of the nodes, for which minimal edit costs are achieved, is denoted as a *minimal mapping* between the two graphs. However, both problems are known to be NP-hard [ZTW+09].

Efficient Change Rule Inference We expect that our envisioned process requires frequent change rule generation since many different API misuse fixes exist. Therefore, a costly change rule inference would hinder its applicability. Thus, we use a simplified heuristic to find the minimal mapping to make change rule inference more efficient. In detail, we construct a *bipartite graph* where one partition consists of the nodes from the misuse AUG (i.e., aug_m) and the other one of the nodes from the fix AUG (i.e., aug_f). Moreover, we balance the number of nodes in both partitions by adding ϵ-nodes in that partition with a lower number of nodes. Note that the nodes within a single partition are not connected to each other, while each single node of one partition is connected to each node of the respective other partition by a weighted edge. The weights represent the *local costs* to transform one node to the respective other, namely, relabeling the node as well as adding, deleting, and relabeling in- and outgoing edges based on AUGs aug_m and aug_f.

Customized Costs: During experimentation, we noticed that equal costs for each change were not effective, namely some updates, such as method relabeling, required a stronger penalty due to their effect on the code behavior. Therefore, we customized the cost function as follows:

- *costs* 1 for adding, deleting, and relabeling edges
- *costs* 2 for adding or deleting a node
- *costs* 4 for relabeling a node

Table 6.1.: Results of the Manual Assessment of the Initial Change Rules

Decision	#Completeness	#Arrangement	#Data
yes	18	14	0
no	32	36	50

The rationale is that nodes usually denote API method calls, which may have a stronger impact on the logic of the program than changing, for instance, the method order. We denote the costs between two nodes stemming from the vertex set of the respective AUGs (i.e., $n_m \in V(aug_m)$ and $n_f \in V(aug_f)$) as $cost(n_m, n_f)$. A complete formal definition of the previous, loosely described cost function can be found in the appendix (cf. Section A.3.1).

Node Matching: Based on this bipartite graph with weighted edges (i.e., by costs), we computed a one-to-one mapping between the nodes of the two partitions achieving overall minimal costs. For this purpose, we applied the Kuhn-Munkres algorithm [Mun57].

Integrating Context In an initial version of the change rule, we considered the change solely *without its context* (cf. Figure 6.2a) [NBKO21a]. In detail, we only selected those node pairs from the minimal mapping *whose weighted edges in the bipartite graph had costs larger than zero*. These pairs denoted the *nodes that required an update*. We constructed the change rules by only keeping these nodes with their respective edges in the AUGs aug_m and aug_f, as well as connecting them according to the minimal mapping with `transform` edges. These form the subgraphs $aug_{m'} \subseteq aug_m$ and $aug_{f'} \subseteq aug_f$, which we mark as *misuse* and *fix* part of the change rule in the graphical representation.

Re-evaluation of Change Rules: When evaluating the effectiveness of these change rules, we found that they miss some essential *context* to fully describe the fix. For this purpose, we analyzed change rules inferred from a random sample of 50 API misuse fixes from the MUBench dataset [ANN+16]. We (i.e., the first three authors of our previous work [NBKO22]) manually assessed these rules using the following three criteria:

- *Completeness:* The change rule contained all necessary API methods (i.e., AUG action nodes) to fix the misuse.
- *Arrangement:* The change rule had a correctly arranged control flow of the necessary API methods (i.e., AUG action nodes) to fix the misuse
- *Data:* The change rule contained all required parameters and `return` values to fix the misuse

These criteria represent the minimal conditions for a valid change rule, and we assessed them individually and independently as *yes*, *no*, or *do not know*. These decisions were also accompanied by a rationale of the decision as well as notes to document further issues. Using Cohen's κ [Coh60] with assessing agreement using the strategy by Landis and Koch [LK77], we obtained $\kappa \approx 0.64$ (i.e., substantial agreement) for *completeness*, $\kappa \approx 0.57$ (i.e., moderate agreement) for arrangement, and $\kappa \approx 0.26$ (i.e., fair agreement) for the data criteria. Therefore, we conservatively summarized the result by only voting the rule criteria as *yes* if all three assessors voted it as *yes* and *no* otherwise. The final results are presented in Table 6.1. We observed that a minority of 18 rules were assessed as *complete*, and only 14 rules were assessed as *correctly arranged*, while none of the rules were evaluated as having the *correct data*. Based on these results, we identified three main challenges:

- **Challenge 1:** Missing data nodes are caused due to non-changed data edges.
- **Challenge 2:** Missing action nodes are caused due to non-changed `finally` edges.
- **Challenge 3:** A huge amount of redundant `order` edges restrict effectiveness and assessment.

Missing Data Nodes: Regarding **Challenge 1**, consider the changes rules in Figure 6.2a and Figure 6.2b. In the former version, we do not know which exact constructor method is called for the `Object.<init>`, which constructs a `QClass` object. It may be that there exist multiple overloaded versions of this constructor. This issue is caused by missing the parameters (i.e., the `BClass` object and `42` constant data nodes) in the rule. The root cause for this problem is that the change only affects the control-flow edges of the constructor node but not its data-flow edges from the incoming data nodes (i.e., its parameters or its receiving objects). Since data nodes are only connected to action nodes via data edges, no change is detected at all, and thus, these nodes were excluded from the rule.

Single-hop-addition: We handled this issue by conducting a post hoc adaption of the change rule after detecting all changed nodes. Particularly, we add all neighbor data nodes of changed action nodes connected via a data-flow edge. We denote this step as *single-hop-addition*. Moreover, for the context of the API usage, it is also important to know how these data nodes are further used. Thus, for each node added by the single-hop-addition, we add all neighbor nodes connected via an outgoing data-flow edge.

Missing Action Nodes: Some change rules missed nodes representing methods called within a `finally` block in Java, causing **Challenge 2**. A `finally` block denotes an enforced default behavior for a `try-catch`-statement even though an exception occurs, for instance, some necessary clean-up operations. The `finally` statements are modeled in AUGs via special `finally` control-flow edges. Similarly to the issue in **Challenge 1**, these edges do not change, and thus, connected nodes are not included in the change rule. We solved this issue by applying the *single-hop addition* also to nodes connected with `finally`-edges.

Many `order` Edges: Finally, in **Challenge 3**, we found too many redundant `order` edges. Recall from Section 3.2.2 that AUGs have many `order` edges due to the conservative strategy of adding them as a transitive closure. These redundant edges, however, cause different problems:

First, `order` edges exist between almost all action nodes, and thus, changing one action node affects many other actions, which causes many non-relevant nodes to be added to the change rule.

Second, many `order` edges are not relevant and blow up the change rule. These large change rules decrease their applicability since computing similarity values becomes more complex and does not scale for many similarity computations.

We reduce the number of `order`-edges by means of *Hsu's algorithm* [Hsu75]. This algorithm computes a *minimal equivalent graph from an existing graph*, namely, a *sub-graph obtained by removing redundant edges so that this graph still keeps the original graph's reachability*. We adapted this algorithm by just removing redundant `order`-edges according to the reachability.

The complete algorithm as pseudocode of ChaRLI can be found in the appendix (cf. Section A.3.1).

6.3.3. Applicability Check

Since different APIs can be misused in various ways, not every change rule may be suitable for each single misuse. Thus, before applying the rule for misuse detection, we checked whether it is *applicable*. We propose two variants of *applicability*, namely, whether

1. the change rule's misuse part is sufficiently similar to the API usage to be analyzed, which we denote as *threshold-based*;
2. the change rule's misuse part is sufficiently similar to known misuses, and its fix part is sufficiently similar to known correct usages, which we denote as *control-group-based*.

Similarity is based on a function sim between two AUGs aug_i and aug_j, computing a normalized similarity, namely $sim(aug_i, aug_j) \in [0, 1]$ where 0 denotes maximum inequality between the AUGs (e.g., completely different code and structure) and 1 indicates that both AUGs are equal. We describe the concrete manifestations of sim afterward.

Threshold-Based For the *threshold-based variant*, we assume a change rule $aug_{m'} \rightarrow aug_{f'}$. Based on this rule, we assess the similarity of the misuse part of a change rule (i.e., $aug_{m'}$) to the actual API usage to compare with (i.e., aug_u). We denote the rule to be applicable if these two AUGs are sufficiently similar based on the similarity function sim. Formally, we define this as:

$$sim(aug_{m'}, aug_u) \geq threshold \tag{6.1}$$

where $threshold \in [0, 1]$ denotes a user-defined hyperparameter. We set this hyperparameter based on our quantitative analysis in Section 6.5.3. In general, this applicability checks how suited the change rule is for the particular API usage.

Control-Group-Based According to the *control-group-based* applicability check, we assume a change rule $aug_{m'} \rightarrow aug_{f'}$ as well as a set of AUGs of known correct usages (i.e., $C = \{aug_{c_1}, aug_{c_2}, \cdots, aug_{c_k}\}$) and misuses (i.e., $M = \{aug_{m_1}, aug_{m_2}, \cdots, aug_{m_l}\}$). We denote a rule to be applicable if the mean similarity between the single parts of the rule (i.e., $aug_{m'}$ and $aug_{f'}$) and AUGs from the sets M and C denotes that for a majority of AUGs in M $aug_{m'}$ is more similar to them than $aug_{f'}$ and for a majority of AUGs in C $aug_{f'}$ is more similar to them than $aug_{m'}$. Formally, this applicability check is described by satisfying the following four conditions based on the sets M and C:

$$\begin{aligned}
\frac{1}{|C|} \sum_{aug_{c_i} \in C} sim(aug_{f'}, aug_{c_i}) &\geq \frac{1}{|M|} \sum_{aug_{m_j} \in M} sim(aug_{f'}, aug_{m_j}) \\
\frac{1}{|C|} \sum_{aug_{c_i} \in C} sim(aug_{m'}, aug_{c_i}) &\leq \frac{1}{|M|} \sum_{aug_{m_j} \in M} sim(aug_{m'}, aug_{m_j}) \\
\frac{1}{|C|} \sum_{aug_{c_i} \in C} sim(aug_{f'}, aug_{c_i}) &\geq \frac{1}{|C|} \sum_{aug_{c_i} \in C} sim(aug_{m'}, aug_{c_i}) \\
\frac{1}{|M|} \sum_{aug_{m_j} \in M} sim(aug_{f'}, aug_{m_j}) &\leq \frac{1}{|M|} \sum_{aug_{m_j} \in M} sim(aug_{m'}, aug_{m_j})
\end{aligned} \tag{6.2}$$

Note that this check requires the availability of valid ground truth of misuses (i.e., M) and correct usages (i.e., C). In general, this criterion denotes how robust a change rule is according to this ground truth.

In our experiments, we will mainly apply Equation 6.1 while we compare both variants (i.e., Equation 6.1 and Equation 6.2) in Section 6.5.4.

6.3.4. Graph Similarity-Based API Misuse Detection

If a change rule $aug_{m'} \rightarrow aug_{f'}$ was applicable to an API usage aug_u, we checked whether the respective API usage is more similar to the misuse part of the change rule than to the fixed part. We formalized this by denoting aug_u as *API misuse* if the following condition holds:

$$sim(aug_{m'}, aug_u) > sim(aug_{f'}, aug_u) \tag{6.3}$$

For validation purposes, we check the decision made by multiple inferred change rules against a ground truth set of API usages and misuses. Thus, we need to define a *correct decision* of a single rule. Particularly, we denote a decision of a rule as *true positive* (i.e., tp) if a known misuse is identified as misuse and as *false negative* (i.e., fn) if not. Similarly, we denote a decision of a rule as *false positive* (i.e., fp) if a known correct usage is identified as misuses and as *true negative* (i.e., tn) if not.

6.3.5. Measuring Graph Similarity

General Terms A similarity measurement *sim* measures the feature of how alike two objects are to each other, usually on a relative scale where 0 indicates maximal inequality and 1 denotes equality [Ife12]. In our case, we apply *sim* on two input AUGs, aug_i and aug_j. There exist different notions to measure similarity, such as *distance-based*, *feature-based*, or *probability-based* similarity [Ife12].

Using only distance-based similarity, however, restricts the number of possible measurements. The rationale is that a distance function $dist : X \times X \rightarrow [0, 1]$ for possible input AUGs aug_i, aug_j, and aug_k represents a metric, which requires the following properties according to Ó Searcóid [Sea07]:

- positive definiteness: $dist(aug_i, aug_j) \geq 0$
- equality: $dist(aug_i, aug_j) = 0 \iff aug_i = aug_j$
- symmetry: $dist(aug_i, aug_j) = dist(aug_j, aug_i)$
- triangle inequality: $dist(aug_i, aug_j) \leq dist(aug_i, aug_k) + dist(aug_k, aug_j)$

While we can apply distance functions as similarity measurements by $sim(aug_i, aug_j) = 1 - dist(aug_i, aug_j)$, not all similarity measurements represent valid distances (e.g., cosine "distance" based on the cosine similarity[2]). Thus, we used the more abstract similarity measurements than solely distance functions.

For that purpose, we selected and defined a set of different similarity functions [NBKO21a]. We base the definition of the similarity functions on the formal description of an AUG introduced in Section 3.2.2. Moreover, we introduce the similarity functions based on the following two AUGs:

$$\begin{aligned} aug_i &:= (V_i, E_i, \Sigma_{V_i}, \Sigma_{E_i}, s_i, t_i, l_{V_i}, l_{E_i}) \\ aug_j &:= (V_j, E_j, \Sigma_{V_j}, \Sigma_{E_j}, s_j, t_j, l_{V_j}, l_{E_j}) \end{aligned}$$

Recall that V denotes the node set, E the edge multiset, Σ_V and Σ_E finite alphabets of labels for nodes and edges, s and t functions for mapping edges to the **s**ource or **t**arget

[2] cosine "distance" violates the equality and triangle inequality condition

node, and l_V and l_E the respective label functions of nodes and edges. Additionally, the function $type : V \to String$ returns the node type as $String$-value (e.g., `MethodCallNode` or `VariableNode`) and the function $api : V \to String$ returns the API related to the node if present as a fully qualified name (e.g., `java.lang.Object`).

Graph Edit Distance We already introduced the *Graph Edit Distance (GED)* in the paragraph on the *change rule inference*. It denotes the minimal costs to perform edit operations (insert, delete, and update nodes and edges) to transform graph aug_i into aug_j [SF83]. Since the GED is widely used to denote inexact matchings [STFR17], we found it applicable for the similarity computation for misuse detection. The main characteristic of GED is its cost function, which has to be carefully chosen [Ser19]. We consider the costs for a single edit operation on a node or an edge. For a formal definition, assume the nodes $n_i, m_i \in V_i$ and $n_j, m_j \in V_j$ and the respective edges $(n_i, m_i) \in E_i$ and $(n_j, m_j) \in E_j$. Then we define the following cost-functions for replacement (i.e., $cost_r$), deletion (i.e., $cost_d$), and addition (i.e., $cost_a$):

$$
\begin{aligned}
cost_r(n_i, n_j) &= \begin{cases} 0 & \text{if } l_{V_i}(n_i) = l_{V_j}(n_j) \wedge type(n_i) = type(n_j) \\ 1 & \text{if } type(n_i) = type(n_j) \\ 2 & \text{otherwise} \end{cases} \\
cost_d(n_i) &= 2 \\
cost_a(n_j) &= 2 \\
cost_r((n_i, m_i), (n_j, m_j)) &= \begin{cases} 0 & \text{if } l_{E_i}((n_i, m_i)) = l_{E_j}((n_j, m_j)) \\ 2 & \text{otherwise} \end{cases} \\
cost_d((n_i, m_i)) &= 2 \\
cost_a((n_j, m_j)) &= 2
\end{aligned}
$$

Note that in opposite to the node costs, we do not have to consider the difference in the types of edges since this is already depicted by the respective edge label. The GED is a function ged minimizing the costs over all possible edit operations $Ed_{i,j}$ (i.e., all node and edge mappings together with possible additions and deletions) required to transform the aug_i into aug_j, particularly

$$
ged(aug_i, aug_j) = \min_{Ed_{i,j}} \left\{ \begin{array}{l} cost_r(n_i, n_j) + cost_d(n_i) + cost_a(n_j) + \\ cost_r((n_i, m_i), (n_j, m_j)) + cost_d((n_i, m_i)) + cost_a((n_j, m_j)) \end{array} \right\}
$$

We obtain the normalized similarity function sim_{ged} by subtracting the fraction of the ged with the maximal costs for all node and edge editions from 1. Particularly, we define the maximal node costs $mcost_n$ and maximal edge costs $mcost_e$ for single node and edge operations as follows:

$$
\begin{aligned}
mcost_n &= \max_{\forall n_i \in V_i, n_j \in V_j} \{cost_r(n_i, n_j), cost_d(n_i), cost_a(n_j)\} \\
mcost_e &= \max_{\forall e_i \in E_i, e_j \in E_j} \{cost_r(e_i, e_j), cost_d(e_i), cost_a(e_j)\}
\end{aligned}
$$

In our customized cost function, this formula is simplified to $mcost_n = mcost_e = 2$. Thus, we finally define sim_{ged} as follows:

$$
sim_{ged}(aug_i, aug_j) = 1 - \frac{ged(aug_i, aug_j)}{\max(|V_i|, |V_j|) \cdot most_n + \max(|E_i|, |E_j|) \cdot most_e}
$$

An efficient algorithm to compute *ged* was suggested by Abu-Aisheh et al. [AARRM15], applying the A*-algorithm, which applied a depth-first search strategy together with a pruning technique to effectively reduce edit operations with high costs. We denote this version as `AStarGED`[3]. However, the exact computation of the GED is known to be NP-hard [ZTW+09]. Therefore, we also use the heuristic approach of the previously introduced Kuhn-Munkres algorithm [Mun57], also known as the Hungarian algorithm. As mentioned before, it computes the minimal edit costs of nodes in a bipartite graph. In detail, we apply the linear sum assignment implemented in the Python library scipy[4] based on the description by Crouse [Cro16]. Since this heuristic only computes the similarity between the nodes, we set $most_e = 0$. We refer to this version as `HungarianGED`.

Maximum Common Subgraph Another similarity measure is based on the *Maximum Common Subgraph (MCS)*. Its notion is to find the largest common subgraph between aug_i and aug_j. In our use case, we are interested in the size of the non-matched nodes and edges [BS98]. This size can be computed by applying the GED using the following cost functions:

$$
\begin{aligned}
cost_r(n_i, n_j) &= \begin{cases} 0 & \text{if } l_{V_i}(n_i) = l_{V_j}(n_j) \wedge type(n_i) = type(n_j) \\ \infty & \text{otherwise} \end{cases} \\
cost_d &= 1 \\
cost_a &= 1 \\
cost_r((n_i, m_i), (n_j, m_j)) &= \begin{cases} 0 & \text{if } l_{E_i}((n_i, m_i)) = l_{E_j}((n_j, m_j)) \\ \infty & \text{otherwise} \end{cases} \\
cost_d((n_i, m_i)) &= 1 \\
cost_a((n_j, m_j)) &= 1
\end{aligned}
$$

This way, we compute the size of the graph parts not belonging to the MCS denoted by the function *mcs* defined by the *ged*:

$$mcs(aug_i, aug_j) = ged(aug_i, aug_j)$$

The normalized similarity function sim_{mcs} is then computed as follows:

$$sim_{mcs}(aug_i, aug_j) = 1 - \frac{mcs(aug_i, aug_j)}{\max(|V_i|, |V_j|) + \max(|E_i|, |E_j|)}$$

Once again, since computing the MCS is NP-hard, we apply the Hungarian algorithm, which, similarly to the GED, computes the MCS among the nodes. Thus, we remove the term $\max(|E_i|, |E_j|)$ from the denominator of the sim_{mcs} since this heuristic does not consider the edge costs. We refer to this version as `HungarianMCS`.

Node-Node-Similarity As a third similarity measurement, we propose the *Node-Node Similarity*. This measure stems from the domain of hyperlinked environments such as the World Wide Web and describes a link-based similarity [Kle98]. Usually, these similarities are computed for nodes from a single graph. However, Blondel et al. [BGH+04] introduced an iterative approach to compute a node similarity for comparing two graphs. In detail,

[3] in [NBKO21a] we referred to this algorithm as `NetworkXGED`

[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html last accessed: 2023/08/14

this variant computes a similarity matrix S as the limit of normalized iterative matrix multiplication as follows:

$$\begin{aligned} S_{k+1} &= A_j S_k A_i^T + A_j^T S_k A_i \\ S &= \lim_{k \to \infty} S_k \end{aligned}$$

The initial matrix S_0 is a matrix of ones, and A_i and A_j denote the adjacency matrices of the AUGs aug_i and aug_j, respectively. Note that in practice, we stopped the iteration if an upper bound of k is reached. Then, we sum over the set of those node pairs that maximized the linear sum assignment denoted as $lsa(S)$. We compute the similarity $sim_{NodeSim}$ by normalizing the similarity as follows

$$sim_{NodeSim}(aug_i, aug_j) = \frac{\sum lsa(S)}{|lsa(S)|}$$

Note that this similarity only relies on structural differences in the graph rather than the label differences. We refer to it as `NodeSimilarityOpt`.

Exas Vectors As a final similarity measurement, we introduce different variants of *Exas vector*-based similarity. Exas vectors describe a graph vectorization technique by Nguyen at al. [NNP+09a] applied in the domain of code clone detection. In their work, they proved that the vector norm of the difference of two Exas vectors is a reasonable approximation of the GED between their represented graphs and, thus, a valid approximation of a graph similarity. The graph vectorization technique characterizes a graph by a set of features present in a graph. In detail, they define (p, q)-nodes and n-paths. (p, q)-nodes describe individual nodes in a graph denoted by their respective labeling function l_V and the number of incoming (i.e., p) and outgoing edges (i.e., q). For instance, in the misuse graph in Figure 6.2b (cf. page 147), one (p, q)-node feature is `Object.<init>-2-1`, ignoring the outgoing `transform`-edge. The n-paths describe a sequence of nodes of length n For instance, in the same graph, there exists the 3-path `Object.<init>` $\xrightarrow{\texttt{def}}$ `QClass` $\xrightarrow{\texttt{recv}}$ `QClass.addData()`. Then, the Exas vector represents the absolute frequencies of all (p, q)-node and n-path features present in a single graph. According to the considerations by Nguyen et al. [NNP+09a], we limit the n-paths to a maximal value of $n = 4$, which obtained sufficient accuracy for their related code clone detection experiments. Moreover, we ignore 1-paths since these represent single-node features already included in the (p, q)-node features. This way, we further reduced the number of different features [NBKO21a, NBKO22].

Vector-Space: We denote the Exas vectors of aug_i and aug_j as vec_i and vec_j, respectively. The difference between the graphs is then represented based on the vector difference between vec_i and vec_j. In theory, vec_i and vec_j would have a countably infinite number of features since both vectors have to be in the same vector space. We handle this by defining the vectors $v\underline{e}c_i$ and $v\underline{e}c_j$ containing only those features that have a count of at least one in both vectors vec_i and vec_j (i.e., cutset of present features). Further, we define the length of an Exas vector as a function len, which returns the number of features with a frequency entry of least one. Our similarity consideres two aspects: first, the similarity of present features (i.e., similar (p, q)-nodes and n-paths) denoted as $feature_{sim}$, and second, the similarity of frequencies of matched features denoted as $featureCount_{sim}$. The $feature_{sim}$ is computed as follows:

$$feature_{sim}(vec_i, vecj) = \max\left(\frac{len(v\underline{e}c_i)}{len(vec_j)}, \frac{len(v\underline{e}c_j)}{len(vec_j)}\right)$$

L1-Norm: For the $featureCount_{sim}$ we applied two different norms, namely the L1-norm computed as follows

$$featureCount_{simL1}(vec_i, vec_j) = 1 - \left|\left| \frac{v\underline{e}c_i - v\underline{e}c_j}{\max(1, maxVal(v\underline{e}c_i - v\underline{e}c_j))} \right|\right|_1$$

The function $maxVal$ computes the maximum absolute value within a vector.

Cosine: Another variant applies the cosine similarity, namely

$$featureCount_{simCos}(vec_i, vec_j) = \frac{\langle v\underline{e}c_i, v\underline{e}c_j \rangle}{||v\underline{e}c_i||_2 \cdot ||v\underline{e}c_j||_2}$$

$\langle \cdot, \cdot \rangle$ computes the scalar product of two vectors.

Exas Vector-Similarity: Finally, we obtain the similarity sim_{Exas} using both aspects $feature_{dist}$ and $featureCount_{dist}$ by applying

$$\begin{aligned} sim_{Exas}(aug_i, aug_j) = \quad & \lambda \cdot feature_{sim}(vec_i, vec_j) + \\ & (1 - \lambda) \cdot featureCount_{sim}(vec_i, vec_j) \\ \lambda \in [0, 1] & \end{aligned} \tag{6.4}$$

The scaling factor λ denotes a hyperparameter, which we set to $\lambda = 0.5$ to obtain equal influence from both similarity flavors (i.e., feature and frequency similarity). When applying $featureCount_{simL1}$, we mark the similarity function as sim_{ExasL1}, and as $sim_{ExasCos}$ when using $featureCount_{simCos}$. We denoted the variants as `ExasVectorL1Norm` and `ExasVectorCosine`, respectively.

We further revised these similarity measurements by identifying and handling observed issues [NBKO21a, NBKO22] as discussed subsequently.

`Indicator-`: First, we found that highly frequent features tend to diminish other feature counts when normalizing the vectors. Therefore, we find it more valuable to simply compare how many features match instead of their frequencies. In detail, we ignore $featureCount_{sim}$ by setting the scaling factor $\lambda = 1$. We mark these variants with the prefix `Indicator-`.

`API-`: Second, we aim to detect misuses of APIs. However, we observed that the similarity was mainly influenced by differences in non-API-specific features in the vectors. Therefore, we suggest only adding features that contain at least one node referring to an API. For that purpose, we applied the function $api : V \to String$ and denoted a node to refer to an API if the api function returned a non-empty string. We mark these variants with the prefix `API-`.

`-Split-`: Third, we observed that complex API usage could intertwine multiple APIs, which hindered the similarity to the noise of potentially non-related APIs. Therefore, we suggest a splitting mechanism for an AUG based on the respective API of the single nodes. In detail, we split the original AUG into several sub-AUGs based on the package of the respective API. Particularly, we use the api function to extract the class name (e.g., `java.lang.Object`) and apply up to the first three elements of the package name reduced by the class name to cluster them as sub-AUGs (e.g., `java.lang`). The nodes that do not relate to an API are collected in a special miscellaneous sub-AUG. Then, we computed an Exas vector for each subgraph indexed by their related package name. Afterward, we compute the similarity between two AUGs by calculating the similarity of each paired sub-AUG matched via their index package name. The overall similarity is the average of the single non-distinct (i.e., $sim \neq 0$) similarity values. We mark these variants with the infix `-Split-`.

Table 6.2.: All different similarity measurements applied for applicability check and similarity-based Misuse Detection together whether these represent distance-based similarities and whether it is a heuristic computation of the similarity function.

#	Similarity	Similarity function	Distance-based	Heuristic Computation
1	`AStarGED`	sim_{ged}	✓	✗
2	`HungarianGED`	sim_{ged}	✓	✓
3	`HungarianMCS`	sim_{mcs}	✓	✓
4	`NodeSimilarityOpt`	$sim_{NodeSim}$	not stated	✓
5	`ExasVectorL1Norm`	sim_{ExasL1}	✓	✗
6	`ExasVectorCosine`	$sim_{ExasCos}$	✗	✗
7	`APIExasVectorL1Norm`	sim_{ExasL1}	✓	✗
8	`APIExasVectorCosine`	$sim_{ExasCos}$	✗	✗
9	`ExasVectorSplitL1Norm`	sim_{ExasL1}	✓	✗
10	`ExasVectorSplitCosine`	$sim_{ExasCos}$	✗	✗
11	`APIExasVectorSplitL1Norm`	sim_{ExasL1}	✓	✗
12	`APIExasVectorSplitCosine`	$sim_{ExasCos}$	✗	✗
13	`IndicatorExasVector`	sim_{Exas}	✓	✗
14	`IndicatorExasVectorSplit`	sim_{Exas}	✓	✗
15	`APIIndicatorExasVector`	sim_{Exas}	✓	✗
16	`APIIndicatorExasVectorSplit`	sim_{Exas}	✓	✗

Combinations: Finally, we also apply *all different combinations* of these variants. In theory, this sums up to 16 different variants, namely, two different norms (i.e., L1 vs. cosine), two different count types (i.e., `Indicator-` or not), two different node types (i.e., `API-` or not), and two different split types (i.e., `-Split-` or not). However, when applying the `Indicator`-variant, using different norms is not necessary. In these cases, we remove the suffix `-L1Norm` and `-Cosine` from the variant's name. Thus, four of the eight combinations applying the `Indicator`-variant can be ignored, summing up to 12 different variants.

We summarize all the different similarity measurements in Table 6.2 denoting whether these are based on a distance function and whether they apply a heuristic.

6.4. Experimental Data and Processing

We evaluated the misuse detection based on the three independent datasets, *MUBench* [ANN+16] and *AU500* [KL21] and our own dataset *AndroidCompass* [NBKO21b]. We already introduced MUBench and AU500 in previous experiments, but we shortly explain the selection process for this setting. Moreover, we present AndroidCompass and its extension named *AndroidCompass+*.

```java
import android.os.Build;
import android.widget.Button;
//[...]
public class AndroidTest {
    //[...]
    public void handleButton(Button button){
        //[...]
        if(Build.VERSION.SDK_INT >= 15){
            button.callOnClick();
        } else {
            button.performClick();
        }
        //[...]
    }
    //[...]
}
```

Code Listing 6.2: Example of an Android Compatibility Check for the `android.widget.Button` API

6.4.1. API Misuse Datasets

MUBench In this setting, the number of misuses in MUBench had increased since we conducted these experiments later. In detail, it consists of 280 API misuses[5]. However, in the evaluation, we only selected 116 of those misuses that were linked to a git repository, provided a commit hash to a fixing commit, and pinpointed to the source file path and the method declaration containing the misuse. This information was necessary to create a valid change rule. Since we were not able to check out three entries (i.e., an entry from the `AspectJ` and two of the `battleforge` project), the overall number of analyzed misuses is 113.

AU500 The AU500 dataset consists of 500 different API usages either labeled as *misuse* (i.e., 115 entries) or *correct usage* (i.e., 385 entries). Next to this label, each entry is linked to its repository, commit hash, source file path, method declaration name, and line number.

AndroidCompass and AndroidCompass+ ***AndroidCompass:*** The AndroidCompass dataset[6] consists of Android compatibility checks, which we collected from apps of FOSS projects from the F-Droid website[7] [NBKO21b]. Especially the Android framework yields many different operating systems and smartphone hardware, as well as a variance of deployed Android versions. Thus, the app's code is vulnerable to incompatibilities with its embedded environment. Compatibility checks describe in-code conditions[8] to determine the API versions or hardware environments (e.g., available sensors) and to prevent the code from calling non-available functionality. Otherwise, functional problems or issues with the performance, security, and user experience can arise [WLC16, HLW+18, LBWK18, WLC19, SBLV+20, WLC+20]. Thus, we interpreted incompatibilities as API misuses. We present an example of an Android compatibility check in Code Listing 6.2. In detail, we observe

[5] as of 2022/01/27
[6] available at https://doi.org/10.5281/zenodo.4428339 last accessed: 2025/02/06
[7] https://f-droid.org/en/ last accessed: 2023/09/01
[8] note that it is also possible to set the Android levels within the configuration [NBKO21b]

a compatibility check in line 8, which verifies whether the Android version[9] used is equal or larger to level `15` (i.e., version code `ICE_CREAM_SANDWICH_MR1`). In this case, we can safely call the `callOnClick()` method based on the documentation[10]. Having an older version, we may use the `performClick()` method instead. The AndroidCompass dataset contains changes to such compatibility checks.

In *AndroidCompass*, we detected whether these checks were added, deleted, or updated and automatically collected 80,324 changed checks together with git repository, commit hash, source file path, changed source line, the comparison type (i.e., `<`,`<=`,`==`,`>=`,`>`), and the Android level to compare with (e.g., `ICE_CREAM_SANDWICH_MR1`). We assumed that the pre-changed version of the code (i.e., before the commit) denoted a false, missing, or incomplete compatibility check and, therefore, an API misuse, while the post-fixed version denoted its respective fix. Since not all changes necessarily target misuses, we filtered the entries in the subsequent *AndroidCompass+* dataset.

AndroidCompass+: In detail, with *AndroidCompass+*, we only considered added or altered `if`-conditions, which protect possibly non-available API method calls. Thus, the change in the condition was more likely due to a non-compatible method call. Moreover, we avoided situations in which incompatibilities were handled by just removing the incompatibility check together with the incompatible method call. Thus, we detailed this original dataset by considering compatibility checks, which handle `if`-conditions with their protected methods in the respective branches (e.g., `callOnClick()` in the `then`- and `performClick()` in the `else`-branch).

For this purpose, we only used entries from AndroidCompass, which did not represent complete added or deleted source files and duplicated files, which reduced the set to 27,873 entries (i.e., changed source files containing compatibility checks). Moreover, we were only able to download 14,225 source file pairs (i.e., the version before and after the changed compatibility check) due to non-available files in the repositories[11].

From these source file pairs, we selected their changed method declarations, for which we could generate change rules using ChaRLI (cf. Section 6.3.2), which was possible for 12,264 single method declarations. Based on an extension of the technique *Relevant API Information Extractor (RAIX)* (cf. Section 5.3.2), which traditionally extracts the containing file, `class`, and method declaration, we parsed and statically analyzed the fixed code to obtain

- the changed `if`-condition (e.g., `Build.VERSION.SDK_INT >= 15`);
- the method calls protected by this condition together with their number of parameters (e.g., as `callOnClick__0` and `performClick__0`);
- and their related branch (i.e., `then` or `else`).

We referred to the method calls in these branches as *protected method calls* by the compatibility check. For instance, for the compatibility check in line 8 in the method declaration `handleButton(Button)` in Code Listing 6.2, we extracted the two protected method calls `callOnClick__0` in the `then` branch and `performClick__0` in the `else` branch.

[9] also known as SDK level (cf. https://developer.android.com/tools/releases/platforms last accessed: 2025/02/06)

[10] https://developer.android.com/reference/android/view/View#callOnClick() last accessed: 2023/08/24

[11] AndroidCompass was obtained in January 2021 while the AndroidCompass+ was obtained in August and September 2024

By using this procedure, we found 2,037 different method declarations changing compatibility checks in `if`-conditions and 12,273 different entries of protected method calls handled by these checks.

Since many different protected method calls can exist, we further reduced them by

a) the most frequently protected ones;

b) those that belong to the Android framework and were not available in any Android version (i.e., SDK level).

By using a), we obtained sufficient examples of conducting API usage pattern mining as well as applying these patterns and the change rules for misuse detection. Using b), we guaranteed that the protected methods referred to the Android API (i.e., representing an API misuse) and that there is evidence that those method calls require a compatibility check (i.e., for those versions, in which they were not available). For a), we selected those methods with at least ten different change entries in the dataset, obtaining 125 different method calls. Regarding b), we further manually checked these method calls whether they refer to the Android framework API (i.e., the methods were declared in a class of the `android.*`-package) and whether these method calls were added in a later Android version (i.e., SDK level > 1) and/or were deprecated from a certain version on. For this purpose, we consulted the official Android API documentation[12]. We selected 36 protected Android method calls, which refer to 1,018 of those 2,037 different method declarations and 1,317 of those 12,273 different protected method calls.

6.4.2. Experimental Settings

Precision and Recall Similarly to the experiments in Chapter 5 and the discussion in Section 5.2.1, we measured the performance of different setting using the precision and the recall. Note that we had specify the measurement of the precision due to the applicability check (cf. *Variants of Applicability Checks*).

MUBench Rules on MUBench Usages For the first setting, we interpreted the pre-fix version of an entry in MUBench as misuse and the post-fix version as correct usage. For each such code pair, we generated the change rule. Then, we validated whether this change rule could classify other pre-fix and post-fix versions in MUBench correctly as misuse and correct usage, respectively. Note that we did not validate the rule on the method where it stemmed from. However, when inspecting the MUBench dataset, we already noticed that there existed many similar API usages from the `jodatime` project. Thus, the results may be positively biased through this setting, meaning it more likely represents an intra-project setting. We refer to this setting as *MUBench-on-MUBench*.

MUBench Rules on AU500 Usages In the second setting, we applied all rules obtained from the fixes of the MUBench dataset on the API usages in the AU500 dataset. Since the projects of MUBench and AU500 are disjunctive, this represents an interproject setting. We refer to it as *MUBench-on-AU500*.

[12] https://developer.android.com/reference/packages last accessed: 2025/02/06

Patterns and Change Rules in AndroidCompass+ The third setup solely compared the pattern-based detector MUDetect [ANN+19b] with RuDetect.

Using MUDetect, we first had to mine patterns from API usages. For this purpose, we built 36 clusters of fixed source files (i.e., those after the changed compatibility check) according to the previously determined 36 protected method calls. In detail, this means that each source file in a cluster contains a compatibility check in an `if`-condition protecting the related protected method call. During mining, we filtered patterns using relative support values. To avoid frequent re-mining, we applied the same set of patterns, and thus including the ground truth fix, for later misuse detection in the related cluster. This way, however, we had to guarantee that the $min_{support_{abs}} > 1$ since otherwise, the ground truth pattern (i.e., the fixed compatibility check) could be present in the set of patterns.

As discussed previously, the change rules used for RuDetect were already present in the AndroidCompass+ dataset.

For both detection variants, we set the ground truth misuses (i.e., $\#tp$ and $\#fn$) as those code versions in the respective cluster *before* the changed compatibility check. Vice versa, we say the ground truth correct usages (i.e., $\#tn$ and $\#fp$) are the code versions in the cluster *after* the changed compatibility check. We refer to this setting as *AndroidCompass+*

Variants of Similarity Measurements We applied the 16 different similarity measurements as demonstrated in the previous section (cf. Table 6.2).

Variants of the Change Rules In our experiments, we analyzed the impact of enhancing the change rules by using the context of the misuse change. In detail, we compared the impact on the API misuse detection using change rules *with* and *without context*, as exemplified in Figure 6.2.

Variants of Applicability Checks We experimented with two different variants of the applicability checks, namely *threshold-based* (cf. Equation 6.1) and *control-group-based* (cf. Equation 6.2). These checks effectively reduced the number of change rules used for a single API usage. In some cases, this led to the case that no rule was denoted as applicable for a usage, and thus, the rules produced no positive results (i.e., $\#tp + \#fp = 0$) at all. Then, we could not compute the precision. We resolved this by only measuring and averaging the precision of rules, producing positive results, which we denoted as *relative precision*. However, we could hardly compare these relative precision using statistic tests since they originate from different numbers of examples. Therefore, we estimated a lower bound of the precision by setting all rules for which we did not obtain any positive result to zero, which we denoted *conservative precision*. We considered this precision as conservative since these rules producing neither *tp*s nor *fp*s are equalized with those producing only *fp*s.

Variants of Misuse Detectors We compared our technique RuDetect to the previously introduced misuse detector MUDetect [ANN+19b] (cf. Section 5.2.4) selected due to the discussion in Section 5.2.4.

We selected this detector because 1) it applied AUGs and thus worked with Java code, 2) it achieved promising results compared to other misuse detectors, and 3) it required a manageable effort for reuse due to available replication packages.

6.5. Validation of RuDetect

Subsequently, we present the validation of RuDetect and ChaRLI.

6.5.1. Validation of the Applicability of ChaRLI

In this section, we present the results on how many change rules could be inferred using the change rule inference technique. In case change rules could not be inferred, we analyzed the root causes, which indicated potential improvements in the inference. This way, we targeted **RQ D-C**.

Methodology We analyzed the change rule inference using the *context* information (i.e., using the *single-hop addition* as denoted in Section 6.3.2) based on the MUBench dataset and our software artifact ChaRLI. This dataset encompasses 113 misuses, as described previously. Moreover, we manually checked for each non-inferred change rule the root causes for non-production by analyzing the respective repository as well as observing the inference step. This way, we determined possible steps for improvements.

Results When analyzing the subset of 113 misuses from MUBench, our artifact ChaRLI initially produced 87 non-empty change rules. After considering the reasons for non-inferring rules, we noticed that five misuse configurations did not link to a proper URL of GitHub, which we fixed accordingly. Based on this, two additional rules could be generated, and thus, we obtained *89 non-empty change rules* in total ($\frac{89}{113} \approx 78.8\%$ of all considered misuses). Note that using rules *without context* information resulted in the same set of inferred change rules for the same misuse fixes.

We analyzed the root causes of non-inference for the other *24* misuse fixes. *13* of them failed due to *location issues* of the misuses caused by falsely or ambiguously configured misuse description (e.g., false method declaration/source file, which should contain the misuse or multiple possible method declarations). For *eleven cases*, ChaRLI had experienced a timeout (i.e., after 5 minutes per rule generation). Even though we did not analyze the reasons for timeouts in detail, we conjecture based on our experience on the rule inference that the change is too complex, and thus, matching the two versions of AUGs takes too long. Thus, overall, in only eleven cases ($\approx 9.7\%$ of all misuse cases), we could not infer change rules due to rule inference itself. Note that we did not and could not always check how many of the 13 cases with location issues work in case the method location worked properly.

Implications We observed that for a majority of misuse fixes (i.e., $\approx 78.8\%$), change rules could be produced. For those 21.2% of fixes for which no change rule could be inferred, we observed that only for 9.7% of all fixes the change rule inference is directly responsible (i.e., timeouts). Thus, we assess the change rule inference as a reasonable technique for real-world cases.

Insight D-12 (RQ D-R): Applicable Automated Change Rule Inference

We observed that ChaRLI produced change rules for a majority of API misuses (i.e., 78.8% in MUBench), while the non-inference of only 9.7% of all potential change rules could be directly traced to the limitations of ChaRLI.

6.5.2. API Misuse Detection Using Similarity Variants

This section presents the evaluation of applying RuDetect with two different experimental settings (i.e., MUBench-on-MUBench and MUBench-on-AU500) to find the best-performing similarities *sim* and *threshold*s for *threshold*-based applicability check. Particularly, we used the successfully inferred change rules from Section 6.5.1 and applied them to the misuse datasets MUBench and AU500. This subsection targets **RQ D-C**.

Methodology We already presented the variants in the datasets (i.e., MUBench and AU500), in the applicability checks (i.e., different *threshold*s in the *threshold*-based applicability check), and in the similarity measurements (i.e., 16 different similarity measurements in Table 6.2) in Section 6.4, which we applied for this evaluation.

In detail, we applied the 89 change rules from MUBench to detect misuses in the MUBench dataset (i.e., MUBench-on-MUBench) as well as in the AU500 dataset (i.e., MUBench-on-AU500). Note that we could only apply these change rules on those 103 entries from the MUBench dataset, for which we could successfully generate the respective AUGs for the misuse and fixed version. Similarly, we could only generate 494 AUGs for the AU500 dataset (cf. Section 6.4.2). As explained before, both datasets contain correct API usages as well as misuses. In MUBench [ANN+16], this is denoted by the pre- and post-version of the fixing commit, and in AU500, the authors of the original paper [KL21] provided ground truth labels. For each change rule, we checked whether it denoted an API usage as misuse (i.e., *positive*) or correct usage (i.e., *negative*) w.r.t. Equation 6.3. Accordingly, denoting a real misuse as misuse was interpreted as *true positive* (tp), and correct usage as misuse as *false positive* (fp). For *true negative* (tn) and *false negative* (fn), the definitions resolved analogously.

According to the misuse criteria in Equation 6.3, we needed to compute the similarity values between the change rule $aug_{m'} \rightarrow aug_{f'}$ and the API usage aug_u. In detail, we computed all similarity values of all 16 different measurements presented before. Then, we applied the *threshold*-based applicability check (cf. Equation 6.1) by testing different values for *threshold* ranging in the interval $[0, 1]$ with 0.1 steps. Based on whether the rule was considered *applicable* with varying *threshold* and similarity measurement, we computed the number of tp, fp, tn, and fn for each rule. Note that in the case of the MUBench-on-MUBench setting, we did not count the case in which a rule was applied to its own usage. Using these values, we could compute the precision and recall. We already described in Section 6.4.2 (cf. *Variants of the Applicability Checks*) that we distinguish between *conservative* and *relative precision*.

We analyzed single variants (e.g., `ExasVectorCosine` with $threshold = 0.6$) by considering the distribution and means of all assessment values (e.g., tp or *relative precision*). Moreover, we applied the Wilcoxon-Mann-Whitney rank sum test [Kan06, p. 101] to compare whether differences in the *conservative precision* and *recall* of certain variants were significant. We chose this test since it did not require a certain statistical distribution between populations (i.e., certain variants) [Kan06, p. 101]. Note that the statistical comparison of the *relative precision* was not meaningful since the configurations lead to different numbers of applicable rules and thus to differently sized populations to compare. Since we compared multiple configurations, we adjusted the significance level ($\alpha = 0.05$) using the Bonferroni correction [Abd07]. For each significant difference, we computed the effect size using Cliff's δ [HK99, KMB+17]. We assessed the effect size using the guideline summarized and presented by Kitchenham et al. [KMB+17], distinguishing between *small*

(i.e., $\delta \geq 0.112$), *medium* (i.e., $\delta \geq 0.276$), and *large* (i.e., $\delta \geq 0.428$) effect interpretation.

We provide all data and analysis scripts[13].

Results At first glance, we visually present the mean values of the relative and conservative precision as well as the recall for the MUBench dataset (cf. Figure 6.3) and AU500 (cf. Figure 6.4). These figures depict general trends among the different similarity measures. For better visual perception, we only show the characteristic similarities, while we provide the graphs of all similarities and tables with all values in the appendix (cf. SectionA.4). Recall that for the relative precision, we could not compute all values since due to the applicability check some computations produced no positive at all. This effect was compensated by the conservative precision, which, however, considered non-positive results of the misuse detection more pessimistically as false positives.

MUBench-on-MUBench: In Figure 6.3, we observed that apart from `AStarGED` and `NodeSimilarityOpt`, the assessment values (i.e., relative and conservative precision as well as recall) for all similarity measurements can be considered constant in the interval $[0, 0.5]$. Note that due to the lack of positive results, `AStarGED` and `NodeSimilarityOpt` produced no positive results for *threshold*s larger than 0.4 and 0.2, respectively.

For the relative precision (cf. Figure 6.3a), we observe an increase at *threshold* 0.6 for all remaining similarities. The largest mean relative precision values at this point were obtained by `APIIndicatorExasVector` ($\approx 96.6\%$)[14], while the lowest ones were produced by `HungarianGED` ($\approx 45.2\%$)[15]. Subsequently, the values become much more unstable, which was caused by the decreasing number of true positive results.

This observation could be further validated by the recall graph in Figure 6.3c. We observed that the mean value of the recall drastically dropped close to zero between *threshold*s 0.2 and 0.3 for `AStarGED` and `NodeSimilarityOpt`. In contrast, for `HungarianMCS` and `HungarianGED` (first one not depicted), it was always very low (i.e., at most $\approx 1.8 - 2.0\%$). For all other Exas vector similarities, the recall behaved almost similarly. Particularly, the recall dropped at *threshold* 0.6 from values of $\approx 18.0 - 26.4\%$ to $\approx 14.9 - 16.9\%$ and even more drastically at 0.9 (i.e., all similarity measurements to $\leq 1.1\%$). This observation indicated that the increase in the relative precision was at the expense of a loss in the recall. We found the Exas vector-based similarities detected $\approx 15.3 - 17.4$ misuses (i.e., absolute number of true positives) on average at $threshold = 0.6$.

This result was also represented by the conservative precision in Figure 6.3b. The conservative precision for `AStarGED`, `NodeSimilarityOpt`, `HungarianGED`, and `HungarianMCS` (the last one not depicted) behaves similarly to the recall, meaning it was or drastically dropped to zero along with an increasing *threshold* value. This effect was also true for the Exas vector similarities, which slightly decreased at *threshold* 0.6 and drastically dropped at 0.9. Having in mind that the high relative precision was obtained at 0.6, we compared the means of the Exas vector similarities at this *threshold*. We observed two clusters with almost similar conservative precision. The first cluster consisted of `IndicatorExasVector`, `IndicatorExasVectorSplit`, `APIIndicatorExasVector`, and `APIIndicatorExasVectorSplit` (the first two not depicted), the conservative precision ranged from $\approx 57.3 - 63.0\%$.

[13] http://doi.org/10.5281/zenodo.15594600
[14] not depicted but similar `IndicatorExasVector` ($\approx 96.9\%$)
[15] not depicted but similar `HungarianMCS` ($\approx 47.1\%$)

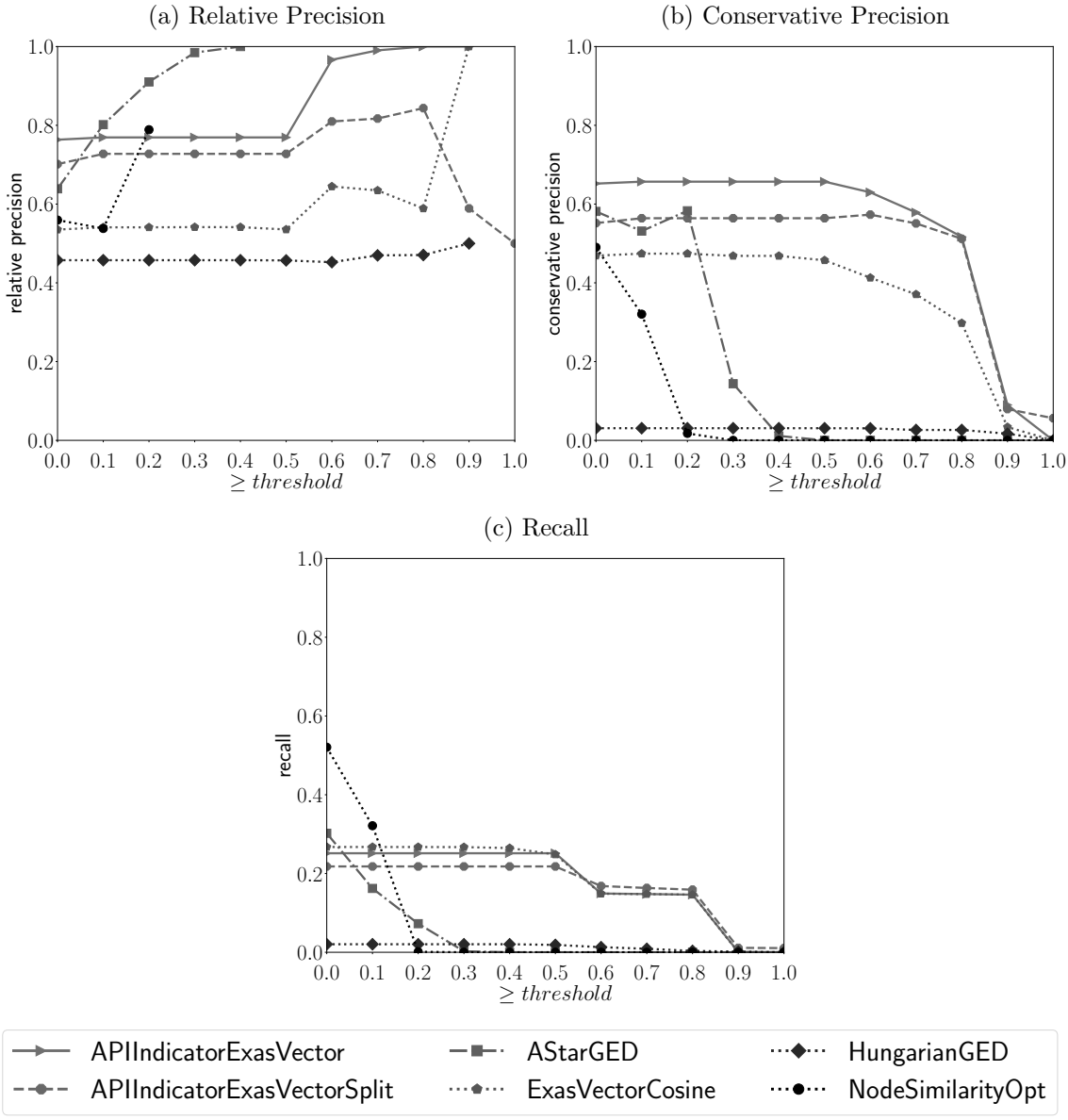


Figure 6.3.: Mean assessment values using different similarity measurements and thresholds for the applicability check in the MUBench-on-MUBench setting.

Table 6.3.: Comparison of different Similarity Measurements for the Conservative Precision of RuDetect in the MUBench-on-MUBench setting at $threshold = 0.6$

	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
APIExasVectorCosine	-	✗	✗	✗	✗	✗	> ✓0.65	✗	✗	✗	✗	> ✓0.62	> ✓0.61	✗	✗	> ✓0.65
APIExasVectorL1Norm	✗	-	✗	✗	< ✓-0.29	✗	> ✓0.64	✗	✗	✗	✗	> ✓0.61	> ✓0.6	< ✓-0.28	< ✓-0.3	> ✓0.64
APIExasVectorSplitCosine	✗	✗	-	✗	< ✓-0.37	< ✓-0.33	> ✓0.69	✗	✗	✗	✗	> ✓0.66	> ✓0.64	< ✓-0.35	< ✓-0.38	> ✓0.69
APIExasVectorSplitL1Norm	✗	✗	✗	-	< ✓-0.37	< ✓-0.33	> ✓0.65	✗	✗	✗	✗	> ✓0.62	> ✓0.6	< ✓-0.35	< ✓-0.38	> ✓0.65
APIIndicatorExasVector	✗	> ✓0.29	> ✓0.37	> ✓0.37	-	✗	> ✓0.65	> ✓0.29	> ✓0.29	> ✓0.34	> ✓0.33	> ✓0.63	> ✓0.63	✗	✗	> ✓0.65
APIIndicatorExasVectorSplit	✗	✗	> ✓0.33	> ✓0.33	✗	-	> ✓0.69	> ✓0.26	> ✓0.25	> ✓0.29	> ✓0.28	> ✓0.66	> ✓0.66	✗	✗	> ✓0.69
AStarGED	< ✓-0.65	< ✓-0.64	< ✓-0.69	< ✓-0.65	< ✓-0.65	< ✓-0.69	-	< ✓-0.64	< ✓-0.64	< ✓-0.72	< ✓-0.7	✗	✗	< ✓-0.64	< ✓-0.73	-
ExasVectorCosine	✗	✗	✗	✗	< ✓-0.29	< ✓-0.26	> ✓0.64	-	✗	✗	✗	> ✓0.61	> ✓0.6	< ✓-0.28	< ✓-0.3	> ✓0.64
ExasVectorL1Norm	✗	✗	✗	✗	< ✓-0.29	< ✓-0.25	> ✓0.64	✗	-	✗	✗	> ✓0.61	> ✓0.6	< ✓-0.27	< ✓-0.3	> ✓0.64
ExasVectorSplitCosine	✗	✗	✗	✗	< ✓-0.34	< ✓-0.29	> ✓0.72	✗	✗	-	✗	> ✓0.69	> ✓0.67	< ✓-0.32	< ✓-0.34	> ✓0.72
ExasVectorSplitL1Norm	✗	✗	✗	✗	< ✓-0.33	< ✓-0.28	> ✓0.7	✗	✗	✗	-	> ✓0.67	> ✓0.65	< ✓-0.31	< ✓-0.33	> ✓0.7
HungarianGED	< ✓-0.62	< ✓-0.61	< ✓-0.66	< ✓-0.62	< ✓-0.63	< ✓-0.66	✗	< ✓-0.61	< ✓-0.61	< ✓-0.69	< ✓-0.67	-	✗	< ✓-0.62	< ✓-0.71	✗
HungarianMCS	< ✓-0.61	< ✓-0.6	< ✓-0.64	< ✓-0.6	< ✓-0.63	< ✓-0.66	✗	< ✓-0.6	< ✓-0.6	< ✓-0.67	< ✓-0.65	✗	-	< ✓-0.62	< ✓-0.71	✗
IndicatorExasVector	✗	> ✓0.28	> ✓0.35	> ✓0.35	✗	✗	> ✓0.64	> ✓0.28	> ✓0.27	> ✓0.32	> ✓0.31	> ✓0.62	> ✓0.62	-	✗	> ✓0.64
IndicatorExasVectorSplit	✗	> ✓0.3	> ✓0.38	> ✓0.38	✗	✗	> ✓0.73	> ✓0.3	> ✓0.3	> ✓0.34	> ✓0.33	> ✓0.71	> ✓0.71	✗	-	> ✓0.73
NodeSimilarityOpt	< ✓-0.65	< ✓-0.64	< ✓-0.69	< ✓-0.65	< ✓-0.65	< ✓-0.69	-	< ✓-0.64	< ✓-0.64	< ✓-0.72	< ✓-0.7	✗	✗	< ✓-0.64	< ✓-0.73	-

Table 6.4.: Comparison of different Similarity Measurements for the Recall of RuDetect in the MUBench-on-MUBench setting at $threshold = 0.6$

	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
APIExasVectorCosine	-	✗	✗	✗	✗	✗	> ✓0.65	✗	✗	✗	✗	> ✓0.59	> ✓0.61	✗	✗	> ✓0.65
APIExasVectorL1Norm	✗	-	✗	✗	✗	✗	> ✓0.64	✗	✗	✗	✗	> ✓0.58	> ✓0.6	✗	< ✓-0.08	> ✓0.64
APIExasVectorSplitCosine	✗	✗	-	> ✓0.03	✗	✗	> ✓0.69	✗	✗	✗	✗	> ✓0.63	> ✓0.65	✗	✗	> ✓0.69
APIExasVectorSplitL1Norm	✗	✗	< ✓-0.03	-	✗	< ✓-0.03	> ✓0.65	✗	✗	✗	✗	> ✓0.59	> ✓0.62	✗	< ✓-0.05	> ✓0.65
APIIndicatorExasVector	✗	✗	✗	✗	-	✗	> ✓0.65	✗	✗	✗	✗	> ✓0.59	> ✓0.61	✗	< ✓-0.08	> ✓0.65
APIIndicatorExasVectorSplit	✗	✗	✗	> ✓0.03	✗	-	> ✓0.69	✗	✗	✗	✗	> ✓0.63	> ✓0.65	✗	✗	> ✓0.69
AStarGED	< ✓-0.65	< ✓-0.64	< ✓-0.69	< ✓-0.65	< ✓-0.65	< ✓-0.69	-	< ✓-0.64	< ✓-0.64	< ✓-0.72	< ✓-0.7	✗	✗	< ✓-0.64	< ✓-0.73	-
ExasVectorCosine	✗	✗	✗	✗	✗	✗	> ✓0.64	-	✗	< ✓-0.08	✗	> ✓0.58	> ✓0.6	-	< ✓-0.09	> ✓0.64
ExasVectorL1Norm	✗	✗	✗	✗	✗	✗	> ✓0.64	✗	-	< ✓-0.08	✗	> ✓0.58	> ✓0.6	✗	< ✓-0.09	> ✓0.64
ExasVectorSplitCosine	✗	✗	✗	✗	✗	✗	> ✓0.72	> ✓0.08	> ✓0.08	-	> ✓0.03	> ✓0.66	> ✓0.69	> ✓0.08	✗	> ✓0.72
ExasVectorSplitL1Norm	✗	✗	✗	✗	✗	✗	> ✓0.7	✗	✗	< ✓-0.03	-	> ✓0.63	> ✓0.66	✗	< ✓-0.04	> ✓0.7
HungarianGED	< ✓-0.59	< ✓-0.58	< ✓-0.63	< ✓-0.59	< ✓-0.59	< ✓-0.63	✗	< ✓-0.58	< ✓-0.58	< ✓-0.66	< ✓-0.63	-	✗	< ✓-0.58	< ✓-0.67	✗
HungarianMCS	< ✓-0.61	< ✓-0.6	< ✓-0.65	< ✓-0.62	< ✓-0.61	< ✓-0.65	✗	< ✓-0.6	< ✓-0.6	< ✓-0.69	< ✓-0.66	✗	-	< ✓-0.6	< ✓-0.7	✗
IndicatorExasVector	✗	✗	✗	✗	✗	✗	> ✓0.64	-	✗	< ✓-0.08	✗	> ✓0.58	> ✓0.6	-	< ✓-0.09	> ✓0.64
IndicatorExasVectorSplit	✗	> ✓0.08	✗	> ✓0.05	> ✓0.08	✗	> ✓0.73	> ✓0.09	> ✓0.09	✗	> ✓0.04	> ✓0.67	> ✓0.7	> ✓0.09	-	> ✓0.73
NodeSimilarityOpt	< ✓-0.65	< ✓-0.64	< ✓-0.69	< ✓-0.65	< ✓-0.65	< ✓-0.69	-	< ✓-0.64	< ✓-0.64	< ✓-0.72	< ✓-0.7	✗	✗	< ✓-0.64	< ✓-0.73	-

The second cluster (i.e., represented by `ExasVectorCosine`) consisted of all other Exas vector similarities, particularly those without the `Indicator-` similarities ranging from $\approx 36.8 - 42.2\%$.

Based on these visual observations, we hypothesized that

1. Exas vector similarities tended to perform better than non-vector-based similarities (i.e., `AStarGED`, `NodeSimilarityOpt`, `HungarianGED`, and `HungarianMCS`) for both conservative precision and recall in the MUBench-on-MUBench setting;
2. `Indicator-`similarities tended to perform best for the conservative precision in the MUBench-on-MUBench setting;

We validated whether these differences are significant by applying the Wilcoxon-Mann-Whitney rank sum test with Bonferroni correction and computed the effect size using Cliff's δ for the conservative precision (cf. Table 6.3) and the recall (cf. Table 6.4).

Using statistical tests for the *conservative precision* in Table 6.3, we confirmed the first hypothesis that `AStarGED`, `NodeSimilarityOpt`, `HungarianGED`, and `HungarianMCS` had significantly lower conservative precision than all Exas vector similarities. All non-Exas vector-based similarities had a large negative effect size compared to the Exas vector ones and thus confirmed the observation from Figure 6.3b. According to the second hypothesis, the statistical test determined significant differences between `Indicator-`based Exas vector similarities and non-`Indicator-`based ones with small to medium effect size. Thus, we could confirm the second hypothesis as well.

Based on the statistical analysis of the recall in Table 6.4, we also confirmed the first conjecture for the recall comparisons. Particularly, all non-Exas vector-based similarities had significantly lower recall than the Exas vector-based ones depicted by a large negative effect size. Note that we also observed some significant differences among the Exas vector similarities (e.g., `ExasVectorSplitCosine` and `ExasVectorCosine`). These results were more sporadically than regularly, and the computed effect sizes were negligible. Thus, we concluded that these differences were a fragment of randomness.

MUBench-on-AU500: In Figure 6.4, we depict the results when applying the rules inferred from MUBench on AU500 with different similarity measures and *threshold*s. Similarly to the MUBench-on-MUBench settting, we observed an almost constant behavior in the *threshold* interval of $[0, 0.5]$ apart from `AStarGED` and `NodeSimilarityOpt`.

Another similarity was the increase in the relative precision (cf. Figure 6.4a) at *threshold* 0.6, using the Exas vector-based similarities. This increase caused high mean values represented by `ExasVectorCosine` ($\approx 92.9\%$) and `APIIndicatorExasVector` ($\approx 92.1\%$) [16]. The `Split`-based Exas vector similarities ranged between $\approx 64.1 - 71.5\%$ at *threshold* 0.6, represented by `APIIndicatorExasVectorSplit` ($\approx 68.6\%$). Non-Exas vector similarities obtained lower (i.e., $< 18\%$ at *threshold* 0.6) or no results at all since no rule had been found to be applicable represented by `AStarGED`, `HungarianGED`, and `NodeSimilarityOpt`.

Again, the high relative precision was accompanied by a drop in the recall (cf. Figure 6.4c). The best-performing similarity measures were `Split`-based Exas vector similarities represented by `APIIndicatorExasVectorSplit` ($\approx 3.6\%$)[17]. This observation indicated a trend towards the `-Split-`similarities but with a very low recall.

[16] similar but not depicted `IndicatorExasVector` ($\approx 92\%$), `ExasVectorL1Norm` ($\approx 91.5\%$), `APIExasVectorCosine` ($\approx 90.9\%$), and `APIExasVectorL1Norm` ($\approx 87.7\%$)

[17] similar but not depicted `IndicatorExasVectorSplit` ($\approx 3.6\%$), `ExasVectorSplitCosine` ($\approx 3.4\%$), and `APIExasVectorSplitCosine` ($\approx 3.4\%$)

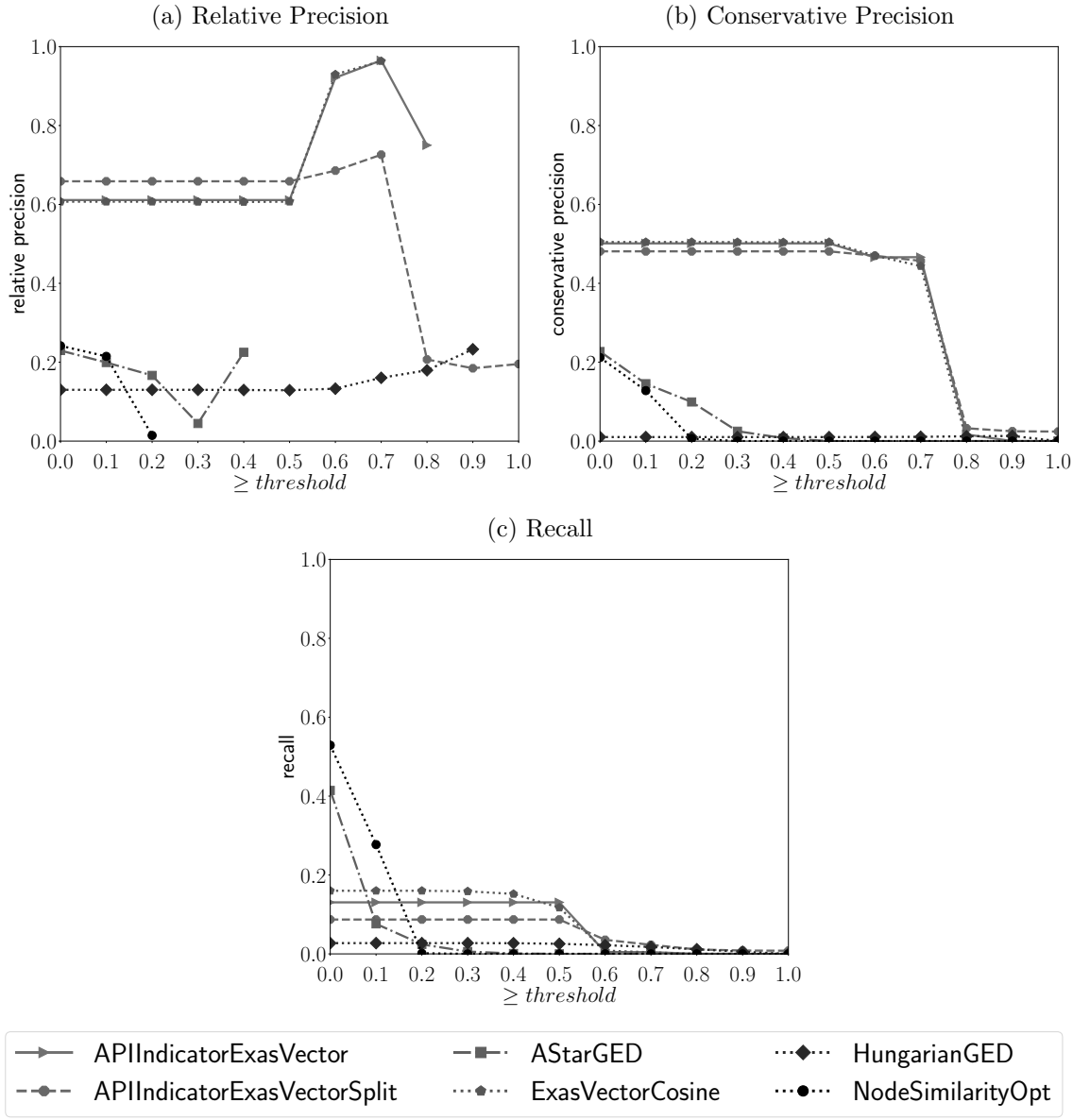


Figure 6.4.: Mean assessment values using different similarity measurements and thresholds for applicability check in the MUBench-on-AU500 setting.

Table 6.5.: Comparison of different Similarity Measurements for the Conservative Precision of RuDetect in the MUBench-on-AU500 setting at $threshold = 0.6$

	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
APIExasVectorCosine	-	✗	✗	✗	✗	✗	> ✓0.51	✗	✗	✗	✗	> ✓0.48	> ✓0.48	✗	✗	> ✓0.51
APIExasVectorL1Norm	✗	-	✗	✗	✗	✗	> ✓0.48	✗	✗	✗	✗	> ✓0.46	> ✓0.46	✗	✗	> ✓0.48
APIExasVectorSplitCosine	✗	✗	-	✗	✗	✗	> ✓0.66	✗	✗	✗	✗	> ✓0.63	> ✓0.63	✗	✗	> ✓0.66
APIExasVectorSplitL1Norm	✗	✗	✗	-	✗	✗	> ✓0.63	✗	✗	✗	✗	> ✓0.6	> ✓0.6	✗	✗	> ✓0.63
APIIndicatorExasVector	✗	✗	✗	✗	-	✗	> ✓0.49	✗	✗	✗	✗	> ✓0.47	> ✓0.47	✗	✗	> ✓0.49
APIIndicatorExasVectorSplit	✗	✗	✗	✗	✗	-	> ✓0.67	✗	✗	✗	✗	> ✓0.64	> ✓0.64	✗	✗	> ✓0.67
AStarGED	< ✓-0.51	< ✓-0.48	< ✓-0.66	< ✓-0.63	< ✓-0.49	< ✓-0.67	-	< ✓-0.51	< ✓-0.48	< ✓-0.65	< ✓-0.62	✗	✗	< ✓-0.55	< ✓-0.7	-
ExasVectorCosine	✗	✗	✗	✗	✗	✗	> ✓0.51	-	✗	✗	✗	> ✓0.48	> ✓0.48	✗	✗	> ✓0.51
ExasVectorL1Norm	✗	✗	✗	✗	✗	✗	> ✓0.48	✗	-	✗	✗	> ✓0.46	> ✓0.46	✗	✗	> ✓0.48
ExasVectorSplitCosine	✗	✗	✗	✗	✗	✗	> ✓0.65	✗	✗	-	✗	> ✓0.62	> ✓0.62	✗	✗	> ✓0.65
ExasVectorSplitL1Norm	✗	✗	✗	✗	✗	✗	> ✓0.62	✗	✗	✗	-	> ✓0.59	> ✓0.59	✗	✗	> ✓0.62
HungarianGED	< ✓-0.48	< ✓-0.46	< ✓-0.63	< ✓-0.6	< ✓-0.47	< ✓-0.64	✗	< ✓-0.48	< ✓-0.46	< ✓-0.62	< ✓-0.59	-	✗	< ✓-0.53	< ✓-0.67	✗
HungarianMCS	< ✓-0.48	< ✓-0.46	< ✓-0.63	< ✓-0.6	< ✓-0.47	< ✓-0.64	✗	< ✓-0.48	< ✓-0.46	< ✓-0.62	< ✓-0.59	✗	-	< ✓-0.53	< ✓-0.67	✗
IndicatorExasVector	✗	✗	✗	✗	✗	✗	> ✓0.55	✗	✗	✗	✗	> ✓0.53	> ✓0.53	-	✗	> ✓0.55
IndicatorExasVectorSplit	✗	✗	✗	✗	✗	✗	> ✓0.7	✗	✗	✗	✗	> ✓0.67	> ✓0.67	✗	-	> ✓0.7
NodeSimilarityOpt	< ✓-0.51	< ✓-0.48	< ✓-0.66	< ✓-0.63	< ✓-0.49	< ✓-0.67	-	< ✓-0.51	< ✓-0.48	< ✓-0.65	< ✓-0.62	✗	✗	< ✓-0.55	< ✓-0.7	-

Table 6.6.: Comparison of different Similarity Measurements for the Recall of RuDetect in the MUBench-on-AU500 setting at *threshold* = 0.6

	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
APIExasVectorCosine	-	✗	< ✓-0.45	< ✓-0.38	✗	< ✓-0.48	> ✓0.51	✗	✗	< ✓-0.42	< ✓-0.37	> ✓0.44	> ✓0.44	✗	< ✓-0.5	> ✓0.51
APIExasVectorL1Norm	✗	-	< ✓-0.48	< ✓-0.41	✗	< ✓-0.51	> ✓0.48	< ✓-0.23	< ✓-0.2	< ✓-0.45	< ✓-0.4	> ✓0.42	> ✓0.42	< ✓-0.28	< ✓-0.53	> ✓0.48
APIExasVectorSplitCosine	> ✓0.45	> ✓0.48	-	✗	> ✓0.44	✗	> ✓0.66	✗	✗	✗	> ✓0.08	> ✓0.59	> ✓0.59	✗	✗	> ✓0.66
APIExasVectorSplitL1Norm	> ✓0.38	> ✓0.41	✗	-	> ✓0.37	< ✓-0.1	> ✓0.63	✗	✗	✗	✗	> ✓0.56	> ✓0.56	✗	✗	> ✓0.63
APIIndicatorExasVector	✗	✗	< ✓-0.44	< ✓-0.37	-	< ✓-0.47	> ✓0.49	< ✓-0.21	< ✓-0.18	< ✓-0.42	< ✓-0.37	> ✓0.43	> ✓0.43	< ✓-0.26	< ✓-0.49	> ✓0.49
APIIndicatorExasVectorSplit	> ✓0.48	> ✓0.51	✗	> ✓0.1	> ✓0.47	-	> ✓0.67	✗	✗	✗	> ✓0.11	> ✓0.6	> ✓0.6	✗	✗	> ✓0.67
AStarGED	< ✓-0.51	< ✓-0.48	< ✓-0.66	< ✓-0.63	< ✓-0.49	< ✓-0.67	-	< ✓-0.51	< ✓-0.48	< ✓-0.65	< ✓-0.62	✗	✗	< ✓-0.55	< ✓-0.7	-
ExasVectorCosine	✗	> ✓0.23	✗	✗	> ✓0.21	✗	> ✓0.51	-	✗	✗	✗	> ✓0.44	> ✓0.44	✗	✗	> ✓0.51
ExasVectorL1Norm	✗	> ✓0.2	✗	✗	> ✓0.18	✗	> ✓0.48	✗	-	✗	✗	> ✓0.42	> ✓0.42	✗	✗	> ✓0.48
ExasVectorSplitCosine	> ✓0.42	> ✓0.45	✗	✗	> ✓0.42	✗	> ✓0.65	✗	✗	-	> ✓0.06	> ✓0.58	> ✓0.58	✗	✗	> ✓0.65
ExasVectorSplitL1Norm	> ✓0.37	> ✓0.4	< ✓-0.08	✗	> ✓0.37	< ✓-0.11	> ✓0.62	✗	✗	< ✓-0.06	-	> ✓0.55	> ✓0.55	✗	< ✓-0.12	> ✓0.62
HungarianGED	< ✓-0.44	< ✓-0.42	< ✓-0.59	< ✓-0.56	< ✓-0.43	< ✓-0.6	✗	< ✓-0.44	< ✓-0.42	< ✓-0.58	< ✓-0.55	-	✗	< ✓-0.48	< ✓-0.62	✗
HungarianMCS	< ✓-0.44	< ✓-0.42	< ✓-0.59	< ✓-0.56	< ✓-0.43	< ✓-0.6	✗	< ✓-0.44	< ✓-0.42	< ✓-0.58	< ✓-0.55	✗	-	< ✓-0.48	< ✓-0.62	✗
IndicatorExasVector	✗	> ✓0.28	✗	✗	> ✓0.26	✗	> ✓0.55	✗	✗	✗	✗	> ✓0.48	> ✓0.48	-	✗	> ✓0.55
IndicatorExasVectorSplit	> ✓0.5	> ✓0.53	✗	✗	> ✓0.49	✗	> ✓0.7	✗	✗	✗	> ✓0.12	> ✓0.62	> ✓0.62	✗	-	> ✓0.7
NodeSimilarityOpt	< ✓-0.51	< ✓-0.48	< ✓-0.66	< ✓-0.63	< ✓-0.49	< ✓-0.67	-	< ✓-0.51	< ✓-0.48	< ✓-0.65	< ✓-0.62	✗	✗	< ✓-0.55	< ✓-0.7	-

Moreover, the low recall indicated that the high precision was the result of only a few correct decisions (i.e., a low number of true positives). We validated the true positives by its mean number for the most precise (i.e., relative precision) similarities, resulting in mean values ranging from $0.5 - 1$ at *threshold* 0.6, and thus, single rules caused a high precision. From Figure 6.4b, we also derived the hypothesis that `AStarGED`, `NodeSimilarityOpt`, `HungarianGED`, and `HungarianMCS` (last one not depicted) performed worse than the Exas vector-based similarities. Among the Exas vector-based similarities, the behavior along an increasing *threshold* was very similar. At $threshold = 0.6$, their conservative precision ranged between 45.3 and 50.7%. Moreover, we observed a massive drop at $threshold = 0.8$, showing similarities to the MUBench-on-MUBench experiment at *threshold* 0.9 for the conservative precision (cf. Figure 6.3b).

Based on this observation, we hypothesized that *Exas vector-based similarity performed better according to the precision than non-Exas vector-based similarities at threshold* $= 0.6$. In contrast to the MUBench-on-MUBench setting, we could hardly conjecture differences in the recall among the similarity measures.

We depict the results of statistical tests for the conservative precision in Table 6.5 and for the recall in Table 6.6. According to the conservative precision, the results strongly confirmed our conjecture that non-Exas vector-based similarities performed worse than Exas vector-based ones. All differences in Table 6.5 between these two groups were significant and had a large negative effect size. Among the Exas vector-based similarities, no significant differences were found.

Regarding the recall, we observed two results. First, `AStar`, `NodeSimilarityOpt`, `HungarianGED`, and `HungarianMCS` had a significantly lower recall than all other Exas vector-based similarity measures with a medium to large negative effect size. Second, the `-Split-`similarity measures had a significantly higher recall than the non`-Split-`variants (except `ExasVectorCosine`, `ExasVectorL1Norm`, and `IndicatorExasVector`) with medium to large effect sizes. This second result was not obvious in Figure 6.4c as the effect size was not measured between the mean values but rather between the different ranking performances of the similarity measures. Taking this into account, we had to conclude that the effect on the mean values by the `-Split-`similarities was negligible.

Implications In summary, we found that

1. there was a sweet spot between the (relative) precision and the recall around the $threshold = 0.6$ using the *threshold*-based applicability check;
2. in all experiments, the Exas vector-based similarities performed better than non-Exas vector-based ones;
3. in the MUBench-on-MUBench experiment, the best-performing similarities applied the `Indicator-`variant, while in the MUBench-on-AU500 experiment, the best-performing ones regarding relative precision used the `Indicator-`option and regarding recall use the `-Split-`option;
4. all similarities currently suffered from a very low recall, indicating that only a few positive examples provided high relative precision.

We further denoted this with the following two insights regarding **RQ D-R**:

Insight D-13 (RQ D-R): Best setup for RuDetect

We found that RuDetect performed best (1) using a threshold-based applicability check at threshold = 0.6, (2) an Exas vector-based similarity with adaptations for specific test setups (i.e., `Indicator-` or `-Split-`)

Insight D-14 (RQ D-R): Careful selection of the *threshold* value when using the *threshold*-based applicability check for RuDetect

When selecting the threshold of the threshold-based applicability check, we observed that a too-low threshold typically negatively influenced precision, while a too-high threshold negatively influenced recall.

These results imply that using the advanced similarity technique with Exas vectors introduced in Section 6.3.5 was beneficial for RuDetect. Moreover, using an applicability check has a positive effect on misuse detection, particularly applying a *threshold*-based applicability check at 0.6. Additionally, using the `Indicator-`option, namely, using the information that certain graph substructures are present in the AUG regardless of its frequency, is a profound tool for improved precision during API misuse detection. This result is also in alignment with the ALP technique suggested by Kang and Lo [KL21], who used characteristic subgraphs to improve misuse detection. Moreover, the `-Split-`variant, namely, splitting into API-specific subgraphs, seems to positively influence the generalization of change rules since we observed positive effects on the recall in the MUBench-on-AU500 experiment. However, these effects were very small.

6.5.3. Impact of the Context of the Change Rules on Misuse Detection

In this section, we present the evaluation on testing the impact of adding context to the change rules, particularly by using the *single-hop-addition* as described in Section 6.3.2. In detail, we compared change rules without context (i.e., without applying *single-hop-addition*) to those with context. Thus, we target **RQ D-R**.

Methodology We conducted a similar procedure as described in Section 6.5.2. In detail, we generated change rules *without context*, which was possible for the same 89 change rules inferred from MUBench as described in Section 6.5.1. Afterward, we computed the similarity values using the 16 similarity measurements between these rules and the single usages of MUBench and AU500. Then, we applied the *threshold*-based applicability check using the different *threshold* values from the interval $[0, 1]$ in 0.1 steps. We computed the relative and conservative precision, as well as recall, as described in the previous experiment. We applied the same ground truth notion as discussed previously, namely, the pre-commit version of MUBench was applied as an API misuse while the post-commit version was applied as a correct usage, and for AU500, the labeled data was used as ground truth. After computation, we calculated the difference in the mean assessment values *with context* by those using rules *without context*. This way, we analyzed whether there exist differences among both variants. We validated whether these differences were significant by conducting once more the Wilcoxon-Mann-Whitney rank sum test [Kan06, p. 101] with Bonferroni correction [Abd07] and computed the effect size using Cliff's δ [HK99, KMB+17].

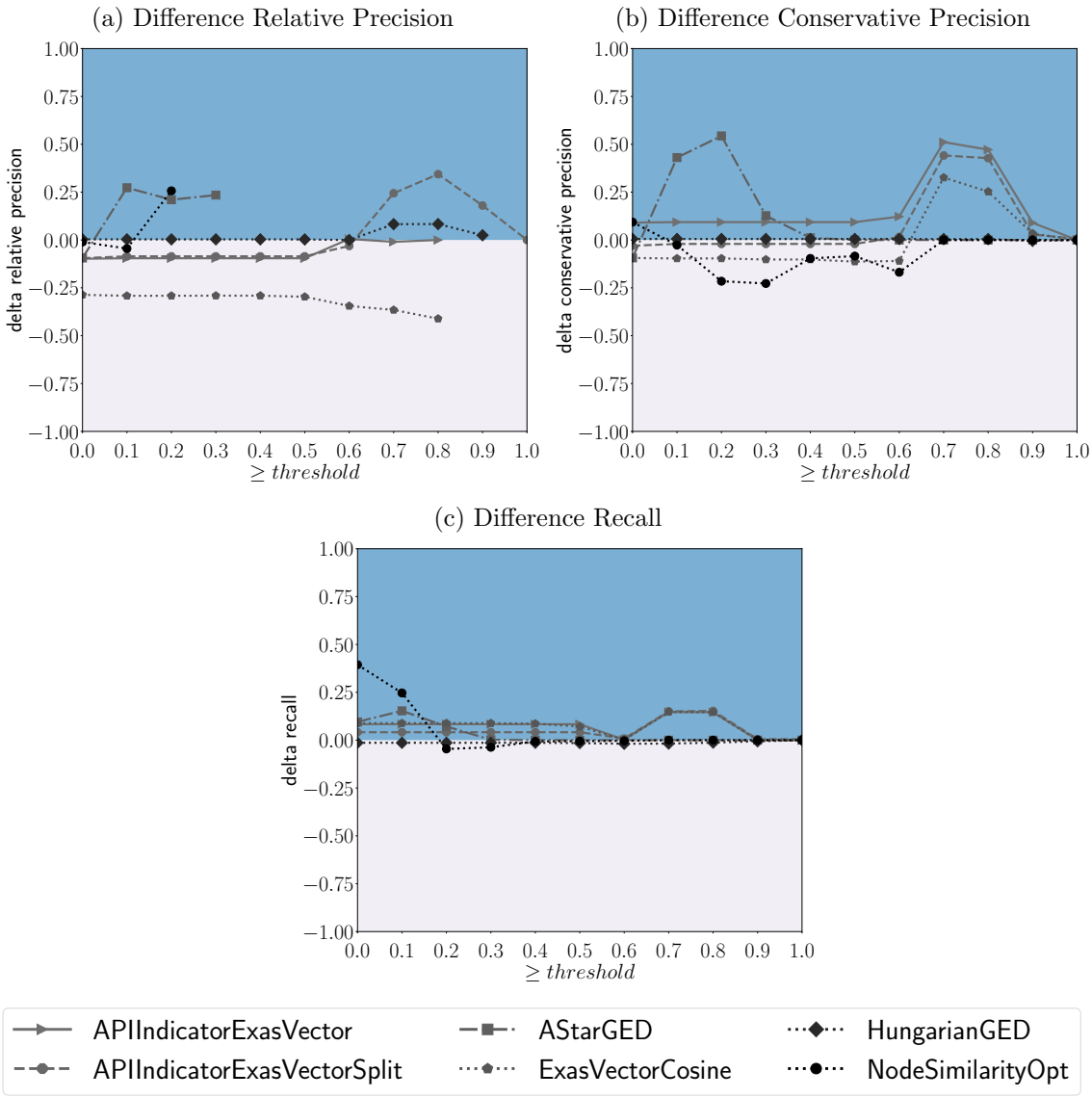


Figure 6.5.: Difference between mean assessment values between rules *with context* subtracted by rules *without context* using different similarity measurements and thresholds for applicability check in the MUBench-on-MUBench setting.

Table 6.7.: Comparison of the conservative precision and the recall between Change Rule *with and without context* at *threshold* = 0.6 in the MUBench-on-MUBench setting. For `AStarGED` no data for testing was available.

With vs. Without Context	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
cons. precision	✗	> ✓0.49	< ✓-0.3	> ✓0.4	✗	✗	-	✗	> ✓0.49	< ✓-0.28	> ✓0.41	✗	✗	✗	✗	< ✓-0.17
recall	✗	> ✓0.59	✗	> ✓0.54	✗	✗	-	✗	> ✓0.58	✗	> ✓0.57	✗	✗	✗	✗	< ✓-0.17

Results We discuss the results separately for the two setting, MUBench-on-MUBench and MUBench-on-AU500, on which the change rules were applied.

MUBench-on-MUBench: We depict the differences in the mean values for *MUBench* in Figure 6.5 for the characteristic similarities. A detailed depiction of all differences and the actual values can be found in the appendix (cf. Section A.4). These figures represent the difference of each assessment value by subtracting the value obtained *with context* from the value obtained *without context* if values for both variants exist. Thus, a positive value (i.e., lying in the upper, blue area of Figure 6.5) indicates a positive effect of the context, while a negative one (i.e., lying in the lower, bright blue area) indicates a negative effect.

When considering the relative precision (cf. Figure 6.5a), we observed that many similarities perform worse *with context* in the *threshold* interval $[0, 0.6]$ than *without*. Two exceptions were `AStarGED` and `NodeSimilarityOpt`, which performed better in the interval $[0.1, 0.3]$. Moreover, some similarities were close to zero, meaning there was almost no difference between the means, such as `HungarianGED` (i.e., for *threshold* ≥ 0.3). We found that almost all Exas vector-based similarities performed equal or worse, ranging in the interval $[0, 0.6]$ between $+0.5\%$ (i.e., `APIIndicatorExasVector` at *threshold* $= 0.6$) to -34.4% (i.e., `ExasVectorCosine` at *threshold* $= 0.6$). For *threshold*s > 0.6, more similarities benefited from the context regarding the relative precision. We analyzed this behavior and found that this was mainly caused by having a small number of applicable rules or none left in the *without context* setting.

When observing the conservative precision (cf. Figure 6.5b), the results were more mixed among the similarity measures. Particularly, in the interval $[0.2, 0.6]$, the results were constant apart from `AStarGED` and `NodeSimilarityOpt`. We observed that some Exas vector-based similarities using L1-norm (i.e., not depicted here) obtain better results through the context ranging from $22.0 - 33\%$, while there were mixed results for `Indicator`-based similarities and worse results for others (at most -19.5%).

Finally, in Figure 6.5c, we observed a mainly positive effect on the mean recall values for almost all similarity measures. The only exceptions were `HungarianGED`, `HungarianMCS` (not depicted), and partially `NodeSimilarityOpt`, having a maximum loss in recall of $\approx 4.6\%$. The other similarities ranged from $0 - 19.8\%$ recall improvement, depending on the *threshold*. At the *threshold* $= 0.6$, the Exas vector-based similarities obtained a recall between $0.2 - 14.8\%$. All non-Exas vector-based variants were below or were slightly negative, being close to zero.

Based on these observations, we hypothesized that

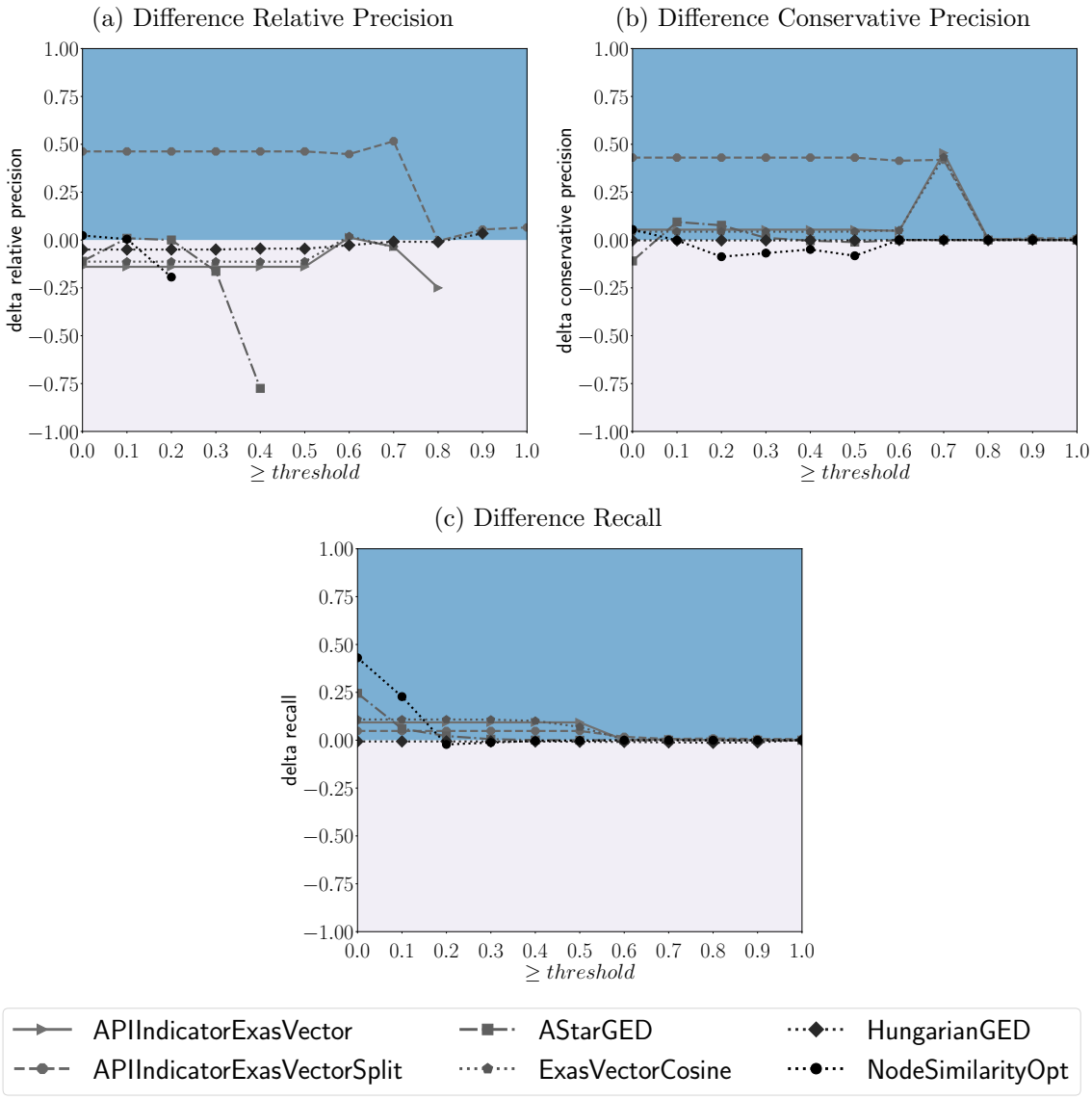


Figure 6.6.: Difference between mean assessment values between rules with subtracted by rules without context using different similarity measurements and thresholds for applicability check in the MUBench-on-AU500 setting.

Table 6.8.: Comparison of the conservative precision and the recall between Change Rule *with and without context* at *threshold* = 0.6 in the MUBench-on-AU500 setting. For `AStarGED` and `NodeSimilarityOpt` no data for testing was available.

With vs. Without Context	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
cons. precision	✗	✗	> ✓0.56	> ✓0.53	✗	> ✓0.56	-	✗	✗	> ✓0.57	> ✓0.54	✗	✗	✗	> ✓0.59	-
recall	✗	✗	> ✓0.45	> ✓0.42	✗	> ✓0.45	-	> ✓0.23	> ✓0.23	> ✓0.46	> ✓0.44	✗	✗	> ✓0.28	> ✓0.48	-

1. rules *with context* had a mixed effect on the precision compared to rules *without context*
2. rules *with context* had a positive effect on the recall compared to rules *without context*

We validated these conjectures with the Wilcoxon-Mann-Whitney rank sum test together with the effect size using Cliff's δ and depict the results in Table 6.7.

Regarding the *first hypothesis*, we found significantly better conservative precision for rules *with context* for all Exas vectors using the `L1Norm` having a medium to large effect size. `NodeSimilarityOpt` performs worse, which, however, has to be taken with a grain of salt, as the mean value of the absolute number of true positives using rules *without context* for this similarity at *threshold* = 0.6 was very low (i.e., ≈ 0.3). Moreover, we observed a worse performance of the `-SplitCosine` similarities with medium effect size.

According to the *second conjecture*, we could confirm this for the Exas vectors using the `L1Norm`, obtaining a large effect size. Note that we also found a significantly worse recall for `NodeSimilarityOpt`, which, as discussed before, was mainly caused by a very small number of true positive results.

For all other similarities, no significant difference had been found neither in the difference of the conservative precision nor in the difference of the recall.

MUBench-on-AU500: In Figure 6.6, we present the differences when comparing rules *with* or *without context* on the *AU500* dataset based on the characteristic similarities (cf. detailed results in the appendix Section A.4). Visually, the results indicated that rules *with context* performed strongly better or only with a few losses in the precision for certain similarities in the interval $[0, 0.7]$. An exception to this observation was `AStarGED`, with a huge loss at *threshold* = 0.4, after which the *with context* variant did not produce any positive results. Thus, we could not compute the relative precision difference for `AStarGED`. Nevertheless, the high precision for this similarity in the *without context* case was mainly caused by very few positive examples, and for *threshold* > 0.5, there were no positive examples available.

In more detail, in Figure 6.6a, all `-Split`-variants represented by `APIIndicatorExasVectorSplit` had an improved relative precision ranging in the interval $[0, 0.7]$ between 41.4 and 57.3%. Those similarities performing worse in the *with context* case did so in the interval $[0, 0.7]$ up to 15.6% loss in relative precision. For *threshold* ≥ 0.8, only little differences were apparent. This result was also consistent with the drop in the relative precision observed in the previous experiment (cf. Figure 6.4a).

For the conservative precision (cf. Figure 6.6b), we only observed negative mean values

for the similarities `AStarGED`, `HungarianGED`, and `NodeSimilarityOpt`[18]. Apart from this single exception, all Exas vector-based similarities represented by `APIIndicatorExasVector` and `APIIndicatorExasVectorSplit` benefited from rules *with context*, obtaining a gain from 0.4 to 43.9% in the interval $[0, 0.7]$. In general, the `-Split-`variants represented by `APIIndicatorExasVectorSplit` benefited most from context (i.e., between +41.3% and +43.9% in the interval $[0, 0.7]$). Once again, beyond *threshold* ≥ 0.8, no apparent differences were observable.

When considering the recall differences (cf. Figure 6.6c), we observed mainly a positive effect using the context. In detail, all Exas vector-based similarities represented by `APIIndicatorExasVector` and `APIIndicatorExasVectorSplit` had a higher recall *with* than *without* context, ranging up to 10.8%. For non-Exas vector-based similarities represented by `AStarGED`, `HungarianGED`, and `NodeSimilarityOpt`, we observed only small losses up to −2.3%. However, at the *threshold* = 0.6, which typically tended to be a sweet spot between precision and recall, these values dropped (i.e., range between $0.03 - 1.8\%$) for the Exas vector-based similarities.

Thus, we conjectured a positive effect for `-Split-`similarities at *threshold* = 0.6 for the conservative precision, while we could not visually conjecture relevant differences for the recall at this *threshold*.

In Table 6.8, we depict the statistical results that confirmed the large positive effect of the *context* within rules using `-Split-`similarities on the conservative precision. Additionally, the test revealed significant differences in the recall for the `-Split-`similarities having a medium to large effect size. For `ExasVectorCosine`, `ExasVectorL1Norm`, and `IndicatorExasVector`, we found a significant difference with a small to medium effect size. Recall that the effect size was mainly a result of the ranking and not the actual difference in the means. Having this in mind, we concluded that the effect on the mean difference in the recall is small.

Implications In summary, we found that

1. using context had mixed effects on the precision depending on the dataset and the applied similarity measurement;

2. using context had a mostly positive effect on the recall even though its effect was not large regarding the mean value.

We summarize this as insight for **RQ D-R**:

Insight D-15 (RQ D-R): Small but positive effect of change rules with context on the recall of RuDetect

*When using change rules with context, we observed a significantly better recall, particularly for the best-performing Exas vector similarities (cf. **Insight D-13**), however, with small effect on the mean recall. We found no significant effect on the precision.*

These results imply a positive effect of using the context on the recall of the misuse detection. Note that this can sound counterintuitive since we expect that change rules *with context* were larger in terms of nodes and edges and thus more specific. A more specific rule, however, should hardly cover more cases in opposition to what is observed by

[18] also negative but not depicted: `HungarianMCS` and `APIExasVectorCosine` at 0.8

a larger recall. A potential explanation for this phenomenon is the similarity measurement, which can cope with larger rules. Particularly, the Exas vector-based similarities measure not only the *shared* features but also incorporate the size of *all present* features in both compared graphs (cf. $feature_{sim}$ in Equation 6.4). That means a larger change rule can cause an increase of the similarity $sim(aug_{m'}, aug_u)$ between an AUG aug_u and its related change rule $aug_{m'} \rightarrow aug_{f'}$. In combination with the *threshold-based applicability check* (cf. Equation 6.1), more change rules become *applicable* and, thus, increase the chance of finding more misuses.

Having in mind the previously observed low recall, having a tool to improve recall without severely harming precision is beneficial. However, the context tends to have a small effect considering the mean recall. Moreover, we also observed that having an applicability check as well as a powerful similarity measurement was a necessary requirement to achieve a good performance.

6.5.4. Impact of the Applicability Check on Misuse Detection

This section presents the results of applying the different variants of *applicability checks* presented in Section 6.3.3. Particularly, we compared the *threshold*-based check (cf. Equation 6.1) with the *control-group*-based check (cf. Equation 6.2).

Methodology Additionally to the computation of the assessment values (i.e., relative and conservative precision as well as recall) using the *threshold*-based applicability check (i.e., Equation 6.1), we computed these values also using the *control-group-based applicability check* (i.e., Equation 6.2). Particularly, we used the MUBench dataset [ANN+16] as the control group since we could derive a set of known misuses (i.e., pre-commit source code) and correct usages (i.e., post-commit source code) from it. Based on this control group, we obtained a set of change rules denoted as *applicable* according to Equation 6.2 and which we applied to the misuse datasets MUBench [ANN+16] and AU500 [KL21] as discussed before as MUBench-on-MUBench and MUBench-on-AU500 settings. This procedure implied using the same 16 similarity metrics using the change rules *with context* only. Since the control-group-based applicability check did not require a *threshold*, we only computed the assessment values once for each similarity (i.e., the means over all applicable change rules depending on the similarity metrics). Based on the previous results, we compared control-group-based applicability only to *threshold*-based applicability at $threshold = 0.6$. The comparison encompassed the *difference in the number of change rules found to be applicable* as well as the assessment values (i.e., relative and conservative precision as well as recall). For the number of applicable change rules, we collected all rules that were applied at least once on the respective misuse dataset (i.e., MUBench and AU500). Similarly to previous experiments, we validated whether the differences in the assessment were significant (i.e., using the Wilcoxon-Mann-Whitney rank sum test with effect size using Cliff's δ).

Results First, we discuss the number of applicable rules found by each applicability check. Note that using the control-group-based check, the number of applicable rules was consistent independently on which misuse dataset these rules were applied for. This characteristic was because the applicability was only decided based on the control group. For the *threshold*-based check, the number of applicable rules varied depending on the misuse dataset since each rule was tested individually on each API usage. We present the results

in Figure 6.7 for applying rules on MUBench (cf. Figure 6.7a) and AU500 (cf. Figure 6.7b). These bar plots depict the proportion of all 89 inferred change rules (cf. Section 6.5.1).

Number Applicable Rules: We observed that in most cases, the *threshold-based check found more applicable rules compared to the control-group-based one*. The only exceptions represented the application of the `AStarGED` and `NodeSimilarityOpt` similarity, for which the *threshold*-based check found no applicable rules at all.

The *threshold*-based check found the largest number of *applicable* rules using `HungarianGED` and `HungarianMCS` for both analyzed misuse datasets. For MUBench, we found that the Exas-vector-based similarities were almost consistent among the `-Split`variants (i.e., $76.4 - 79.8\%$) and slightly less for all other Exas vector-based entries (i.e., $64 - 66.2\%$).

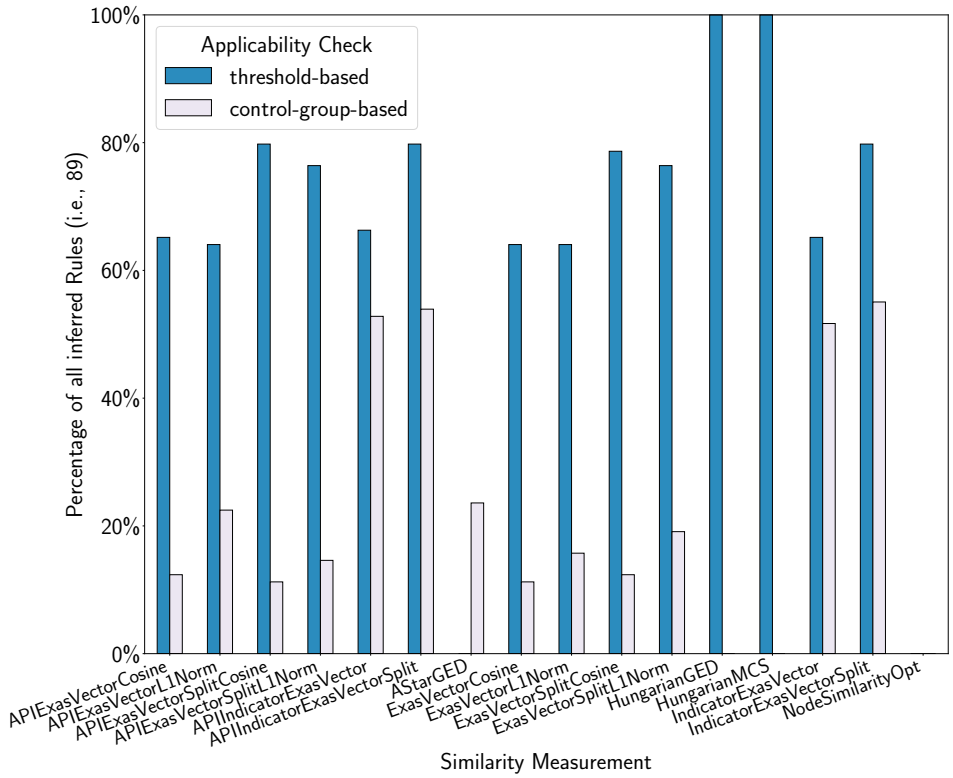
The control-group-based check had the largest proportion of applicable rules using the `Indicator`-Exas-vector similarities (i.e., between 51.7 and 55.1%). For the other similarities, the proportion ranged between 11.2% and 23.6% or obtained no applicable rules at all (i.e., `HungarianGED`, `HungarianMCS`, and `NodeSimilarityOpt`).

Second, we compared the assessment values between both applicability checks by considering the mean values of relative and conservative precision as well as the recall for MUBench-on-MUBench (cf. Figure 6.8) and MUBench-on-AU500 (cf. Figure 6.9). These boxplots summarize the assessment values *among all* similarities. We omitted the detailed view due to visibility reasons. A detailed distinction *between* the similarities can be found in the appendix (cf. Section A.3.2). Note that the *mean* is represented by $\times$ while the *median* is given by a horizontal line within the boxplot.

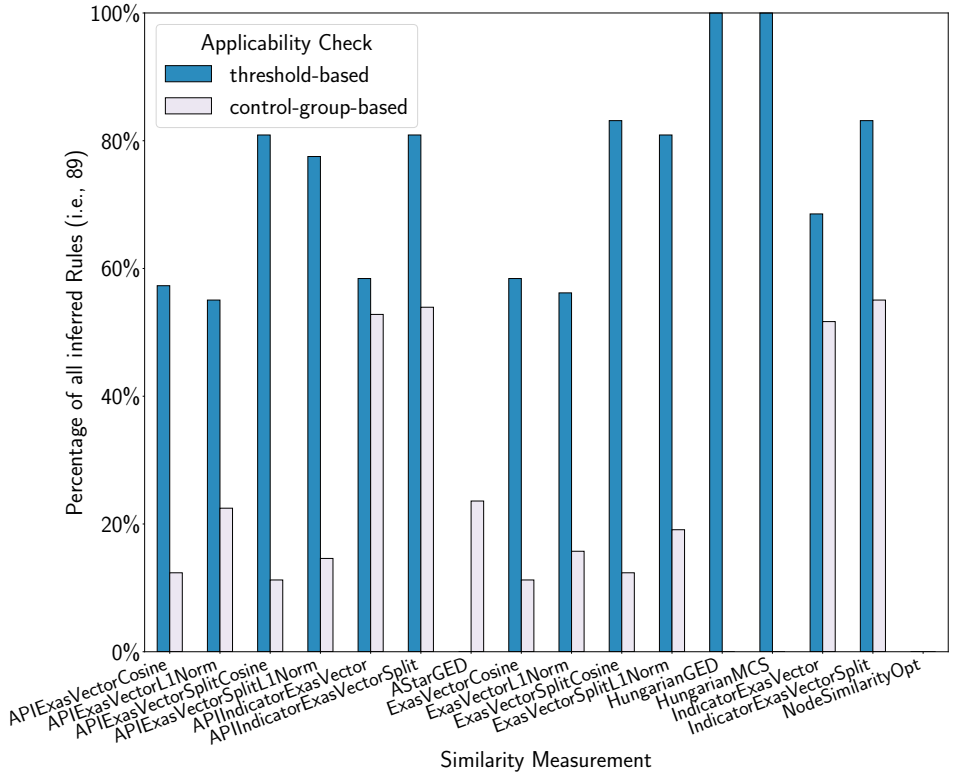
MUBench-on-MUBench: When comparing both checks based on *MUBench*, we observed that the mean value of the relative precision of the control-group-based check (cf. Figure 6.8a) was slightly larger than the one using the *threshold*-based applicability check. Nevertheless, the distribution of the *threshold*-based check had a larger variance, making it hard to conjecture general trends. For conservative precision and recall (cf. Figures 6.8b and 6.8c), the *threshold*-based variant had a larger mean value than the control-group-based check. Even though the *threshold*-based check was once again accompanied by a larger variance and the control-group-based check had some good-performing outliers, we hypothesized *a benefit in conservative precision and recall when applying the threshold-based* due to the difference in the mean values.

MUBench-on-AU500: For *AU500*, we observed almost identical boxplots for both applicability checks for the relative precision (cf. Figure 6.9a). Similar to the results observed in the MUBench-on-MUBench setting, the mean values and distribution of the *threshold*-based were larger for conservative precision and recall compared to its counterpart (cf. Figures 6.9b and 6.9c). Thus, we expected *a positive effect of applying the threshold-based check* for AU500 as well. However, *the effect on the recall seemed to be very small, and thus, we also expected a small effect size.*

Statistical Differences in both Settings: We validated whether the difference in conservative precision and recall were significant (i.e., depicted in Table 6.9 for MUBench-on-MUBench and Table 6.10 for MUBench-on-AU500). In this case, we distinguished among the different similarities individually. For both datasets, we observed a significant difference among the applicability checks except for `HungarianGED` and `HungarianMCS` (i.e., for both datasets) and some `Indicator`-variants. Note that we did not obtain a sufficiently large number of positive results using `NodeSimilarityOpt`. For the remaining similarity measures, in all but two cases, the *threshold*-based had a small to large positive effect on the conservative precision. The only exceptions were found for `AStarGED` with a small

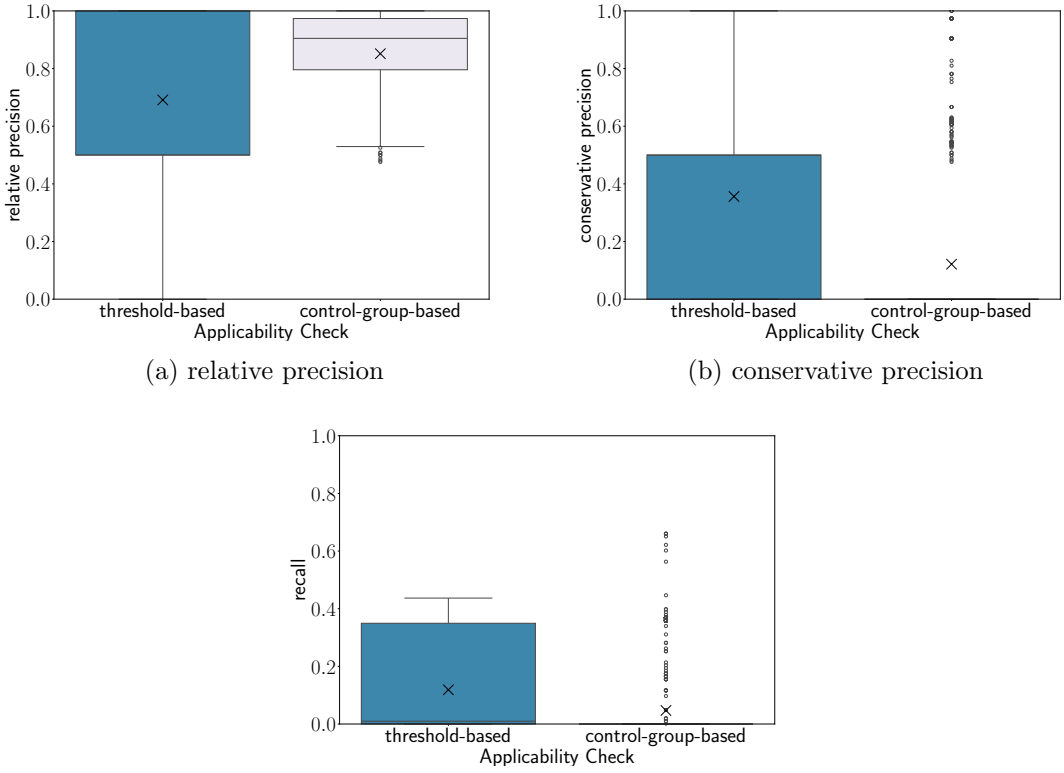


(a) on MUBench



(b) On AU500

Figure 6.7.: Difference in the number of distinct applicable rules using *threshold*-based and control-group-based applicability check



(a) relative precision

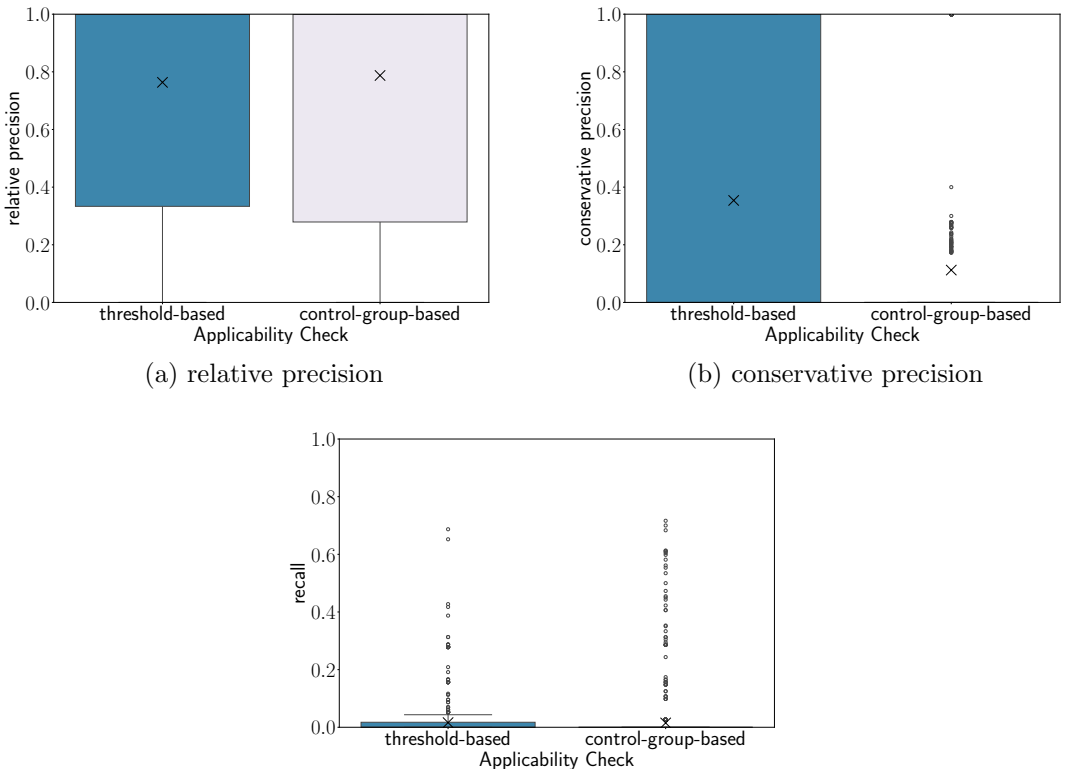
(b) conservative precision

(c) recall

Figure 6.8.: Comparison the assessment values between Change Rules using *threshold*-based applicability check and control-group-based check in MUBench-on-MUBench setting.

Table 6.9.: Comparison of the conservative precision and the recall between Change Rules using *threshold*-based applicability check and control-group-based check in the MUBench-on-MUBench setting. For `NodeSimilarityOpt` no data for testing was available.

threshold **vs. control-group**	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
cons. precision	> ✓0.62	> ✓0.49	> ✓0.69	> ✓0.62	> ✓0.45	> ✓0.34	< ✓-0.24	> ✓0.63	> ✓0.58	> ✓0.7	> ✓0.6	✗	✗	> ✓0.44	> ✓0.39	-
recall	> ✓0.63	> ✓0.54	> ✓0.69	> ✓0.63	✗	✗	< ✓-0.24	> ✓0.63	> ✓0.6	> ✓0.71	> ✓0.63	✗	✗	✗	✗	-



(a) relative precision

(b) conservative precision

(c) recall

Figure 6.9.: Comparison the assessment values between Change Rules using *threshold*-based applicability check and control-group-based check in MUBench-on-AU500 setting.

Table 6.10.: Comparison of the conservative precision and the recall between Change Rules using *threshold*-based applicability check and control-group-based check in the MUBench-on-AU500 setting. For `NodeSimilarityOpt` no data for testing was available.

threshold **vs. control-group**	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
cons. precision	> ✓0.5	> ✓0.42	> ✓0.67	> ✓0.62	✗	> ✓0.15	< ✓-0.15	> ✓0.5	> ✓0.45	> ✓0.65	> ✓0.59	✗	✗	> ✓0.11	> ✓0.15	-
recall	> ✓0.48	✗	> ✓0.67	> ✓0.58	< ✓-0.13	✗	< ✓-0.15	> ✓0.48	> ✓0.37	> ✓0.64	> ✓0.54	✗	✗	✗	✗	-

negative effect and `APIIndicatorExasVector` with no effect in MUBench-on-AU500. The recall of the *threshold*-based variant had mostly a positive effect ranging from small to large effect size depending on the dataset and similarity. Note that in the MUBench-on-MUBench setting, among the Exas-vector-based similarities, we found no significant difference between using `Indicator`-characteristic. For MUBench-on-AU500 setting, this was similar, with the exception of `APIIndicatorExasVector` had a small negative effect using the *threshold*-based variant and `APIExasVectorL1Norm`, for which we also could not determine a significant effect.

Implications All in all, we found that

1. using the *threshold*-based applicability check usually found more *applicable* rules than using the control-group-based check;
2. using the *threshold*-based applicability check usually had a positive effect on the misuse detection with Exas-vector-based similarities in terms of conservative precision and recall than the control-group-based check with exceptions when using `Indicator`-based similarities.

We summarize this as the following insight regarding **RQ D-R**:

Insight D-16 (RQ D-R): Positive effect of the *threshold*-based over the control-group-based applicability check for RuDetect

When testing the influence of applicability checks, we found that threshold-based derived more applicable rules, by which it had a typically larger conservative precision and recall compared to the control-group based applicability check.

Our results show that the *threshold*-based applicability check, which assesses each rule individually based on the respective API usage, is more beneficial than the control-group-based approach. Having in mind that the second check requires a sufficiently large set of misuses and correct usages, we can strengthen the argument towards the *threshold*-based check, which does not require this prerequisite. Nevertheless, we observe that solely using the applicability is not sufficient and requires a good-performing similarity measurement for change rule-based misuse detection, particularly the Exas-vector-based similarity.

6.5.5. Comparison to the State-of-the-Art

Finally, we compared RuDetect to the misuse detector MUDetect [ANN+19b] with the implementation available on GitHub[19] as previously presented (cf. Sections 5.2.4). This comparison targets **RQ D-R**.

Methodology In order to have a valid comparison with RuDetect, we applied MUDetect on the same datasets using the same donor code to infer patterns. For that purpose, we discussed the particular configuration when applying MUDetect in the *MUBench-on-MUBench-*, the *MUBench-on-AU500-*, and in the *AndroidCompass+-*setting (cf. Section 6.4.2). In all three settings, we measured and compared the performance (i.e., precision and recall) of RuDetect and MUDetect.

[19] https://github.com/stg-tud/MUDetect last accessed: 2024/07/16

Moreover, MUDetect, like many pattern-based misuse detectors, selects single patterns via a ranking mechanism. Therefore, we also applied a selection scheme for change rules produced by ChaRLI for the *MUBench-on-MUBench-* and the *MUBench-on-AU500*-settings. This procedure allowed a direct comparison of the obtained precision and recall, while in *AndroidCompass+*, we conducted statistical tests among multiple patterns and change rules used for detection.

Pattern Mining Procedure: *MUDetect* uses an Apriori-based Frequent Pattern Mining (FPM) technique to mine frequent sub-AUGs as patterns whose violations are denoted as API misuses. Since we inferred change rules from the MUBench dataset, we mined patterns from MUBench (i.e., and thus in the MUBench-on-MUBench and MUBench-on-AU500 setting) as well, namely, from the fixed version of the misuse, which we considered as correct usages. The patterns from MUBench were selected using the minimal absolute support $min_{support_{abs}}$ (cf. Section 3.5.2). We collected patterns from all method declarations, and thus AUGs that were present in the respective source file containing the misuse fix (i.e., in MUDetect, this is denoted as *cross-method* [ANN+19b]). Then, we varied the $min_{support_{abs}}$ value from the set $\{5, 10, 20, 40\}$, which we obtained by doubling the minimal support as long as the precision of the misuse detection decreases. To avoid multiple FPM runs among single API usage instances, we inferred the patterns once and applied the violation detection on each API usage afterward. Note that in the MUBench-on-MUBench setting, this meant that the actual fix is part of the FPM, and thus, the patterns were positively biased. We compensated this bias by increasing $min_{support_{abs}}$ by one (i.e., using the set $\{6, 11, 21, 41\}$).

Setting of Pattern-based Misuse Detection: For violation detection, we applied the default violation strategy by MUDetect together with the accompanied filtering of alternative patterns and ranking strategy [ANN+19b]. We determined the number of tp, fp, tn, and fn and computed the *precision and recall* (cf. Equation 5.1 and Equation 5.2). Note that since MUDetect had no mechanism to decide whether a pattern *is applicable* for violation detection, we considered all non-violated patterns as negative results (i.e., either tn or fn), which might negatively bias the recall compared to RuDetect.

Change Rule Selection: For the MUBench-on-MUBench and the MUBench-on-AU500 settings, we applied a selection mechanism for change rules. Recall that we observed in Section 6.5.2 that an increasing *threshold* of the similarity (i.e., $sim(aug_{m'}, aug_u)$) between the misuse part of the rule (i.e., $aug_{m'}$) and the currently analyzed API usage (i.e., aug_u) in the *threshold-based applicability check* (cf. Equation 6.1) had a positive impact on the relative precision. Thus, we conjectured that a change rule selection based on the $sim(aug_{m'}, aug_u)$ was reasonable. Therefore, we selected a change rule for misuse detection by the following two-step approach:

1. Select that change rule (if available) that satisfied the applicability check (cf. Equation 6.1) with $threshold \in \{0.6, 0.7\}$ and which obtained the largest $sim(aug_{m'}, aug_u)$, namely, that rule whose misuse part was most similar to the actual usage.

2. In case multiple rules had the same largest $sim(aug_{m'}, aug_u)$, we decided based on the majority, while ties were decided towards a negative result (i.e., no misuse).

Note that in the MUBench-on-MUBench-setting, we excluded similarities $sim(aug_{m'}, aug_u)$ obtained by changes rules applied to their 'own' misuse.

MUDetect vs. RuDetect: We compared MUDetect against our technique RuDetect using this selection scheme of rules by using the MUBench-on-MUBench as well as the

MUBench-on-AU500 setting. The second one represented a cross-project variant since MUBench and AU500 did not share any common software projects. In both settings, we applied all similarity metrics for RuDetect (cf. Table 6.2).

In the third setting, we compared both detectors using the *AndroidCompass+* dataset. In detail, we conducted the pattern mining on the clusters of fixed source files with protected method calls, namely, those methods protected by a compatibility check (cf. Section 6.4.2). In this experiment, the FPM of MUDetect applied the minimal relative support (cf. Section 3.5.2) with $min_{support} \in \{0.1, 0.2\}$ but ensuring that the absolute support $min_{support_{abs}}$ was larger than 1 to avoid bias of the ground truth (cf. Section 6.4.1). For RuDetect, we also used for the applicability check $threshold \in \{0.6, 0.7\}$ but only applied two similarity metrics, namely, `ExasVectorCosine` and `APIIndicatorExasVector`, both selected due to their performance in the previous MUBench-on-MuBench and MUBench-on-AU500-setting.

Again, we denoted the API usage *before* the changed compatibility check as *misuse* and the version *after* the change as *correct usage*. For MUDetect, we applied both sets of patterns (i.e., obtained with $min_{support}$ 0.1 and 0.2) for misuse detection once again with the default violation strategy and filter and ranking mechanisms [ANN+19b]. For RuDetect, we inferred all change rules and applied them to the misuse (i.e., *before* change) and correct usage (i.e., *after* change) of all AUGs in the related cluster of protected methods except for its own ground truth (i.e., we did not apply the change rule on its own misuse or correct usage). In both variants, we tested if at least one pattern or one change rule detected a misuse (i.e., either a tp or fp depending on the tested AUG), and if none detected a misuse, we denoted this as a negative result (i.e., either a tn or fn depending on the tested AUG).

Once again, we assessed in the AndroidCompass+-setting the significance of different precision and recall by applying the Wilcoxon-Mann-Whitney rank sum test [Kan06] with Bonferroni correction [Abd07] and computed the effect size using Cliff's δ [HK99, KMB+17].

Results We depict the best results (i.e., precision and recall) from all variants of RuDetect and MUDetect of the MUBench-on-MUBench- and MUBench-on-AU500-settings as bar plots in Figures 6.10 and 6.11 as well as all detailed results in Tables 6.11 and 6.12[20].

MUBench-on-MUBench: For the *MUBench-on-MUBench*-setting, we observed in the lower part of Table 6.11 that the most precise result of MUDetect is at a *support* value of 21 with 73.7% with a recall of 49.6%. Based on our selection scheme, RuDetect obtained with two similarities, namely, `APIIndicatorExasVector` and `IndicatorExasVector`, large precision. `APIIndicatorExasVector` obtained 93.9% ($threshold = 0.6$) and 98.2% ($threshold = 0.7$), while `IndicatorExasVector` had a precision of 93.8% ($threshold = 0.6$) and 98.2% ($threshold = 0.7$). This high precision was accompanied by a recall of 54.9% ($threshold = 0.6$) and 49.6% ($threshold = 0.7$) for `APIIndicatorExasVector` as well as 53.1% ($threshold = 0.6$) and 49.6% ($threshold = 0.7$) for `IndicatorExasVector`. Thus, both variants outperformed the best MUDetect result at *support* value 21, having a comparable recall. Note that MUDetect obtained better recall at *support* value 6 (i.e., 64.6%) but with a much worse precision (i.e., 63.5%). In contrast, the `ExasVectorCosine`-variant performed worse than MUDetect, having a precision of $59.4 - 59.6\%$ but with a comparable recall. All other Exas vector-based similarities obtained precision values ranging from $48.1 - 62.1\%$, while all non-Exas vector-based similarities obtained no positive results at all.

[20] A more detailed view, for instance including the absolute number of #tp,#fp,#tn,#fn of RuDetect (cf. Table A.10, Table A.11, Table A.12, and Table A.13) and MUDetect (cf. Table A.9) as well as further tested variants are included in the appendix (cf. Section A.4.1)

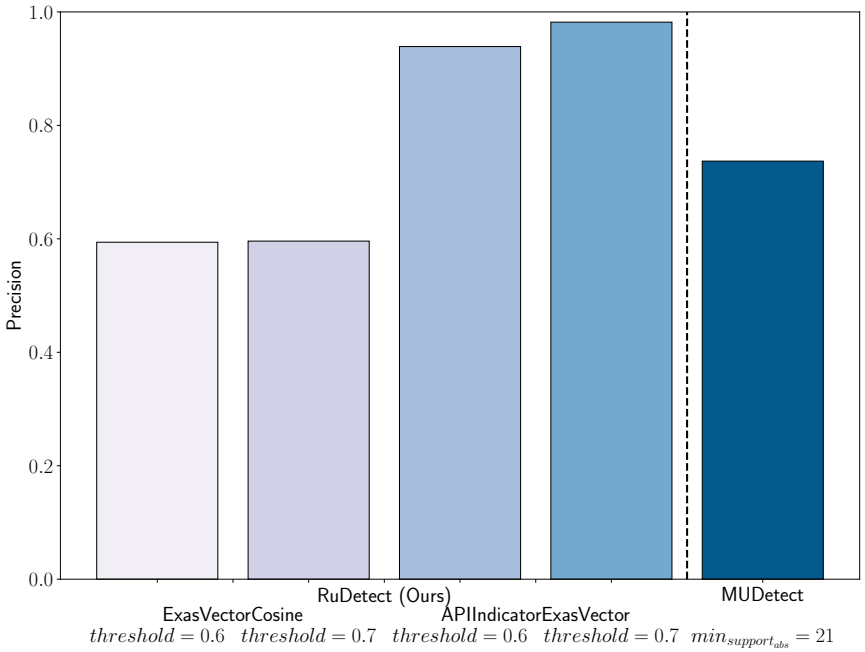
These results align with the results observed in Table 6.3, in which we found significantly better results when using `Indicator`-based similarities.

MUBench-on-AU500: Second, we compared both detectors on the *MUBench-on-AU500 setting* with the results depicted in the lower part of Table 6.12 for MUDetect and the upper part for RuDetect. We found the most precise results of MUDetect at *support*-value 20 with 37.8%, accompanied by a recall of 14.8%. For RuDetect, we observed the most precise result for $threshold = 0.6$ using the `ExasVectorL1Norm` with 45.5%, followed by the `ExasVectorCosine` with 38.5%, both obtaining the same recall of 4.3% at $threshold = 0.6$. With $threshold = 0.7$, both variants had further increased precision, namely, 75% for `ExasVectorCosine` and 66.7% for `ExasVectorL1Norm`. However, both at the expense of losing recall (i.e., 2.6% and 1.7, respectively). Using the `APIIndicatorExasVector` similarity, we obtained a precision between $15.8 - 50\%$ with a low recall of 2.6% as well. Thus, these variants obtained a larger precision than MUDetect but with the downside of a lower recall. All other Exas-vector-based similarities resulted in precision values ranging from $17.6 - 36.4\%$ for $threshold = 0.6$ and $18.9 - 60\%$ for $threshold = 0.7$. Similarly to the MUBench-on-MUBench setting, we did not obtain any positive results for the non-Exas-vector-based similarities.

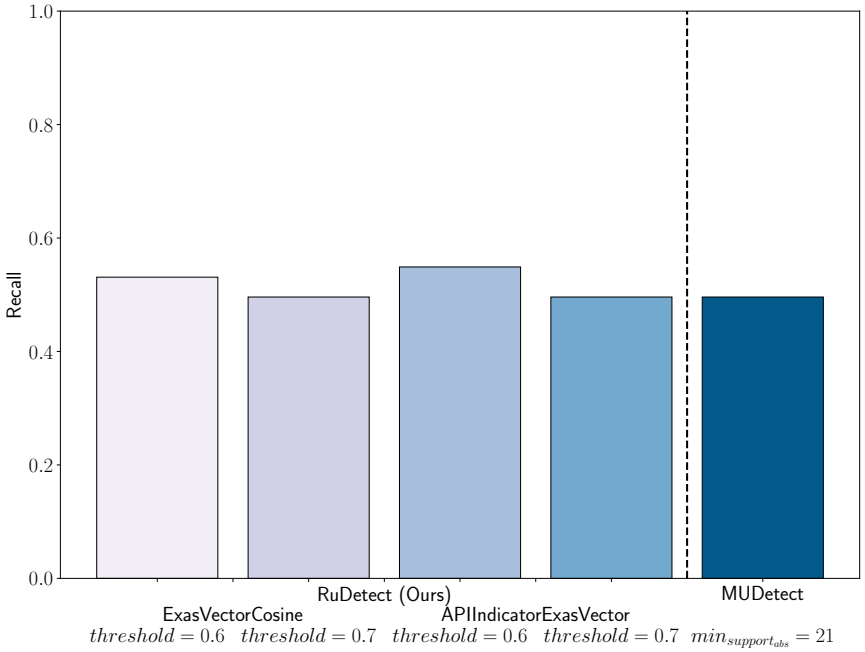
Having these results, we obtained the following insight:

Insight D-17 (RQ D-R): Dependency on Similarities and *threshold* for improved Precision of RuDetect compared to MUDetect

RuDetect obtained partially (i.e., dependent on certain similarities and thresholds) better results in terms of precision compared to MUDetect. In a within-project *setting (i.e., MUBench-on-MUBench), we found the best results using the* ***`APIIndicator-ExasVector`****-similarity, while in a* cross-project *setting (i.e., MUBench-on-AU500), we found the* ***`ExasVectorCosine`****-similarity perform best.*



(a) Precision

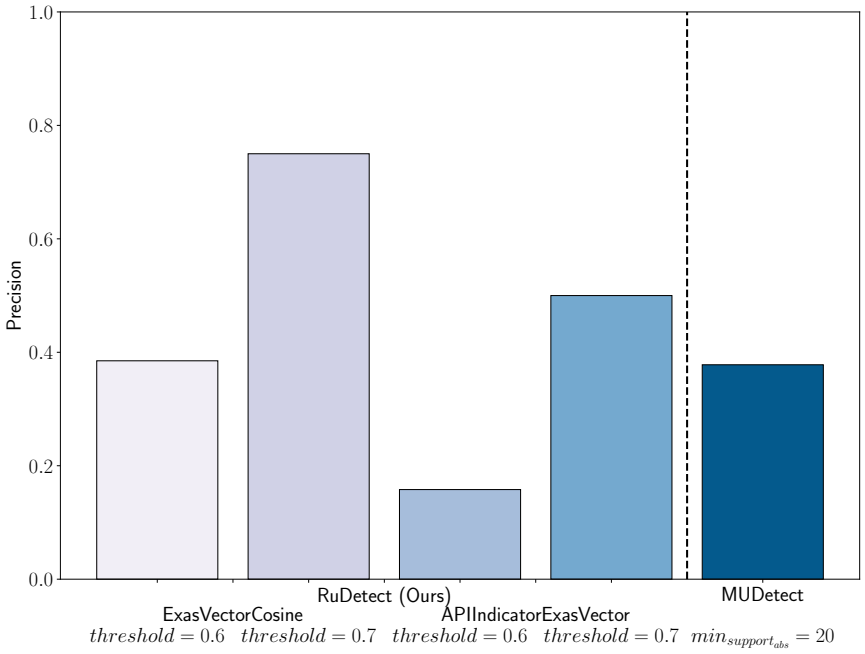


(b) Recall

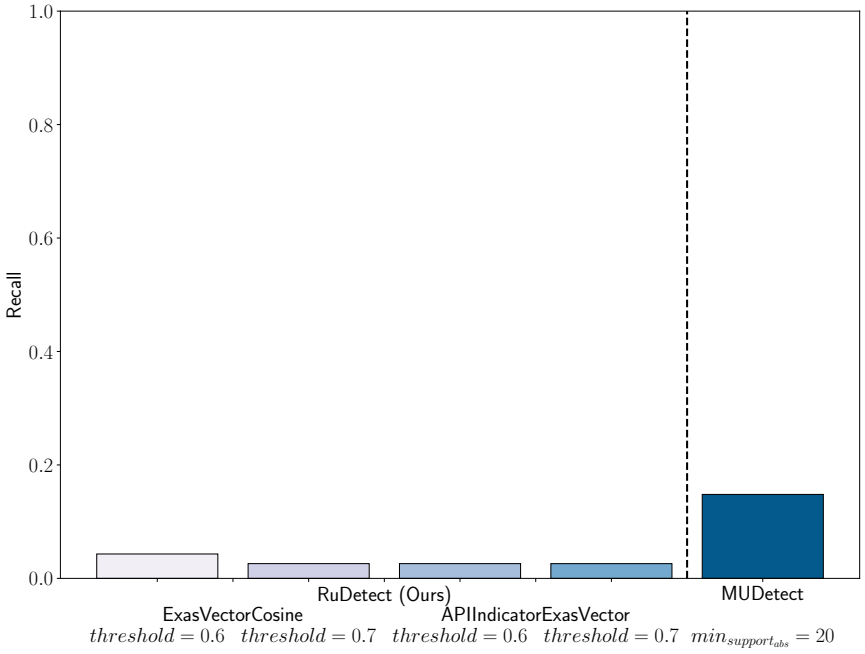
Figure 6.10.: Best results as barplots from different settings of RuDetect and MUDetect int the MUBench-on-MUBench setting

Table 6.11.: Precision and recall of all variants of RuDetect and MUDetect in the MUBench-on-MUBench-setting. Highlighted entries depicted in Figure 6.10

		Precision (%)		**Recall (%)**	
		threshold		*threshold*	
		= 0.6	= 0.7	= 0.6	= 0.7
RuDetect (Ours)	APIExasVectorCosine	60.2	59.6	54.9	49.6
	APIExasVectorL1Norm	60.6	57.3	53.1	45.1
	APIExasVectorSplitCosine	53.5	48.1	20.4	11.5
	APIExasVectorSplitL1Norm	57.1	53.3	17.7	7.1
	APIIndicatorExasVector	93.9	98.2	54.9	49.6
	APIIndicatorExasVectorSplit	52.3	46.4	20.4	11.5
	AStarGED	-	-	0.0	0.0
	ExasVectorCosine	59.4	59.6	53.1	49.6
	ExasVectorL1Norm	59.6	56.3	52.2	43.4
	ExasVectorSplitCosine	61.0	61.3	22.1	16.8
	ExasVectorSplitL1Norm	59.0	60.9	20.4	12.4
	HungarianGED	-	-	0.0	0.0
	HungarianMCS	-	-	0.0	0.0
	IndicatorExasVector	93.8	98.2	53.1	49.6
	IndicatorExasVectorSplit	61.5	62.1	21.2	15.9
	NodeSimilarityOpt	-	-	0.0	0.0
MUDetect	$min_{support_{abs}} = 6$	63.5		64.6	
	$min_{support_{abs}} = 11$	68.2		51.3	
	$min_{support_{abs}} = 21$	73.7		49.6	
	$min_{support_{abs}} = 41$	0.0		0.0	



(a) Precision



(b) Recall

Figure 6.11.: Best results as barplots from different settings of RuDetect and MUDetect on MUBench-on-AU500

Table 6.12.: Precision and recall of all variants of RuDetect and MUDetect in the MUBench-on-AU500-setting. Highlighted entries depicted in Figure 6.11

		Precision (%)		**Recall (%)**	
		threshold		*threshold*	
		= 0.6	= 0.7	= 0.6	= 0.7
RuDetect (Ours)	APIExasVectorCosine	30.8	60.0	3.5	2.6
	APIExasVectorL1Norm	36.4	50.0	3.5	1.7
	APIExasVectorSplitCosine	17.6	20.0	16.5	13.0
	APIExasVectorSplitL1Norm	18.7	18.9	14.8	8.7
	APIIndicatorExasVector	15.8	50.0	2.6	2.6
	APIIndicatorExasVectorSplit	19.0	22.1	17.4	14.8
	AStarGED	-	-	0.0	0.0
	ExasVectorCosine	38.5	75.0	4.3	2.6
	ExasVectorL1Norm	45.5	66.7	4.3	1.7
	ExasVectorSplitCosine	17.8	22.1	18.3	16.5
	ExasVectorSplitL1Norm	18.9	22.6	17.4	12.2
	HungarianGED	-	-	0.0	0.0
	HungarianMCS	-	-	0.0	0.0
	IndicatorExasVector	18.5	42.9	4.3	2.6
	IndicatorExasVectorSplit	19.8	23.9	20.0	18.3
	NodeSimilarityOpt	-	-	0.0	0.0
MUDetect	$min_{support_{abs}} = 5$	19.4		33.0	
	$min_{support_{abs}} = 10$	36.0		15.7	
	$min_{support_{abs}} = 20$	37.8		14.8	
	$min_{support_{abs}} = 40$	37.0		8.7	

Low Recall of RuDetect: We found in the *MUBench-on-AU500*-setting that RuDetect obtained large precision at the expense of low recall. We conjecture that since change rules require an actual edit of a misuse, RuDetect can only detect those misuses that were previously fixed in the training set. In contrast, MUDetect, which inferred patterns from all API usages in the training set, does not require them to be changed. Based on the two settings, we hypothesize that the dissimilarity of the changed APIs in MUBench and in AU500 could cause issues regarding matching change rules to find misuses. We manually assessed this by comparing the change rules inferred from MUBench regarding their possible misuse detection ability on the misuses in AU500. We found that only a minority of rules were, in principle, able to detect the misuse (i.e., only two out of 37 misuses from MUBench used the API class and method present in AU500, and both already used this API correctly). Therefore, we hypothesize that in the AndroidCompass+ dataset, the chance of finding more matching change rules within the pre-defined clusters is higher.

Insight D-18 (RQ D-R): Dependency on the training set for a larger recall of RuDetect compared to MUDetect

RuDetect resulted in a cross-project setting (i.e., MUBench-on-AU500) into a lower recall than MUDetect. Based on a manual analysis, we determined that only a few change rules from MUBench were able to detect misuses in AU500. This strengthens the conjecture that RuDetects' recall is dependent on the training data.

Insight D-19 (RQ D-R): No strong dependency on the training set for larger precision of RuDetect compared to MUDetect

While RuDetect suffered from low recall, it still outperformed MUDetect regarding precision, as seen in both tested experimental settings (i.e., MUBench-on-MUBench and MUBench-on-AU500).

AndroidCompass+: In the third setting, we evaluated the hypothesis of better performance, especially regarding the recall. We applied pattern mining and change rule inference using ChaRLI on the 36 clusters (i.e., based on the method calls protected by an Android compatibility check as discussed in Section 6.4.1) in *AndroidCompass+* using the previously discussed variants (i.e., varying similarity, *threshold*, and $min_{support}$). Afterward, we applied the misuse detectors RuDetect and MUDetect in the above-mentioned setting.

We depict the obtained precision and recall for each cluster in its respective boxplot in Figure 6.12 (i.e., precision) and Figure 6.13 (i.e., recall). In Figure 6.12, we observed that the distribution of the precision values was larger for all variants of RuDetect compared to MUDetect. Nevertheless, the interquartile range of the four variants of RuDetect lay above two variants of MUDetect. This observation was further supported by larger mean values for RuDetect (i.e., $63.3 - 68.6\%$) compared to MUDetect (i.e., $47.3 - 49.1\%$) as well as median values (i.e., $64 - 72\%$ for RuDetect and 50% for MUDetect). Regarding recall (cf. Figure 6.13), a larger difference between the RuDetect and the MUDetect with $min_{support} = 0.2$ was observable (i.e., mean is 22% and median 18.3%). In contrast, the difference between the MUDetect variant using $min_{support} = 0.1$ and the variants from RuDetect was smaller. In detail, this MUDetect variant had a mean recall of 38% and a median recall of 38.7%, while the RuDetect variants' mean ranged from $38.4 - 43.3\%$ and their median from $37.8 - 43\%$.

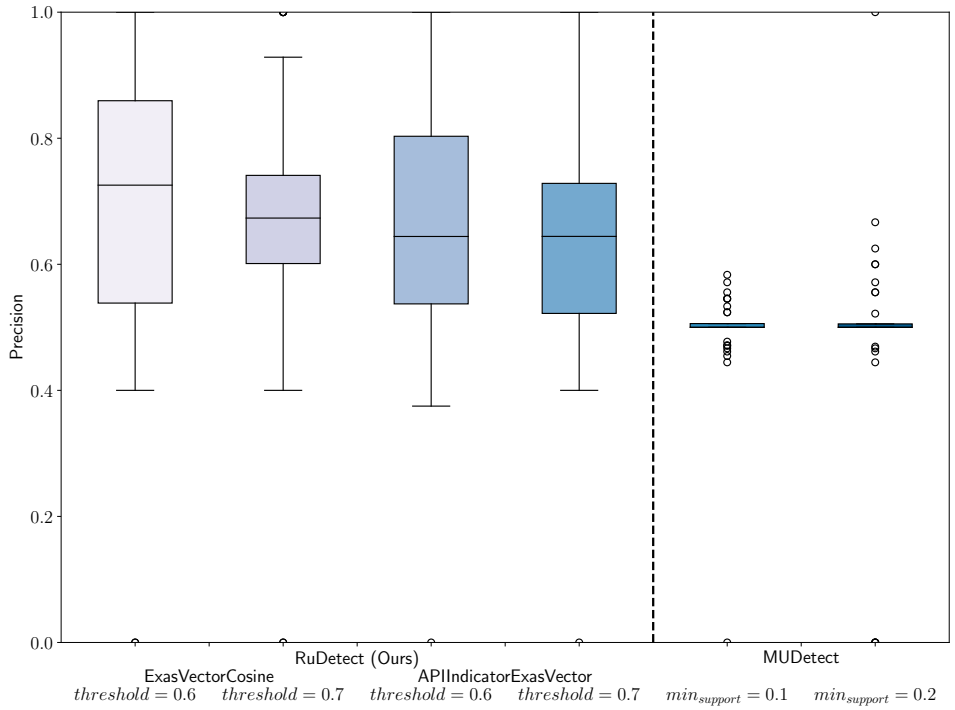


Figure 6.12.: Precision results as boxplot of different settings of RuDetect and MUDetect in the AndroidCompass+ setting

Table 6.13.: Precision statistical test results (Wilcoxon signed-rank test with significant results with effect size using Cliff's δ) between different variants of RuDetect and MUDetect in the AndroidCompass+ setting

			RuDetect (Ours)				MUDetect	
			ExasVectorCosine		APIIndicatorExasVector			
			$threshold = 0.6$	$threshold = 0.7$	$threshold = 0.6$	$threshold = 0.7$	$min_{support} = 0.1$	$min_{support} = 0.2$
RuDetect (Ours)	ExasVector-Cosine	$threshold = 0.6$	-	✗	✗	✗	> ✓0.65	> ✓0.61
		$threshold = 0.7$	✗	-	✗	✗	> ✓0.7	> ✓0.64
	APIIndicator-ExasVector	$threshold = 0.6$	✗	✗	-	✗	> ✓0.62	> ✓0.55
		$threshold = 0.7$	✗	✗	✗	-	> ✓0.56	> ✓0.51
MUDetect		$min_{support} = 0.1$	< ✓-0.65	< ✓-0.7	< ✓-0.62	< ✓-0.56	-	✗
		$min_{support} = 0.2$	< ✓-0.61	< ✓-0.64	< ✓-0.55	< ✓-0.51	✗	-

Table 6.14.: Recall statistical test results (Wilcoxon signed-rank test with significant results with effect size using Cliff's δ) between different variants of RuDetect and MUDetect in the AndroidCompass+ setting

			RuDetect (Ours)				MUDetect	
			ExasVectorCosine		APIIndicatorExasVector			
			$threshold = 0.6$	$threshold = 0.7$	$threshold = 0.6$	$threshold = 0.7$	$min_{support} = 0.1$	$min_{support} = 0.2$
RuDetect (Ours)	ExasVector-Cosine	$threshold = 0.6$	-	✗	✗	✗	✗	> ✓0.54
		$threshold = 0.7$	✗	-	✗	✗	✗	> ✓0.48
	APIIndicator-ExasVector	$threshold = 0.6$	✗	✗	-	✗	✗	> ✓0.56
		$threshold = 0.7$	✗	✗	✗	-	✗	> ✓0.55
MUDetect		$min_{support} = 0.1$	✗	✗	✗	✗	-	> ✓0.49
		$min_{support} = 0.2$	< ✓-0.54	< ✓-0.48	< ✓-0.56	< ✓-0.55	< ✓-0.49	-

We further checked whether the observed differences were significant using the Wilcoxon-Mann-Whitney rank sum test [Kan06] with $\alpha = 0.05$ while applying Bonferroni correction [Abd07] and determined the effect size using Cliff's δ [HK99, KMB+17]. We applied a rank test since we could not directly assume a normal distribution of the precision and recall. The results are presented in Table 6.13 for precision and Table 6.14 for recall. We could approve the visually observed results regarding precision, particularly all variants of RuDetect had *a significantly larger precision* (cf. Table 6.13) than those from MUDetect with *a large effect size* (i.e., according to the assessment of Cliff's δ based on [KMB+17]). All interdifferences of the variants of RuDetect were not significant. Regarding the recall, we determined that all variants of RuDetect and the variant of MUDetect using $min_{support} = 0.1$ were significantly larger than that variant of MUDetect using $min_{support} = 0.2$ (cf. Table 6.14). Similarly to the precision, all other interdifferences were not significant.

Insight D-20 (RQ D-R): RuDetect significantly performs better than MUDetect

Having AndroidCompass+ *as an improved training setup compared to the* MUBench-on-AU500*, we determined that RuDetect significantly performed better (independent of the considered similarity and threshold) than MUDetect regarding the precision while obtaining a comparable to better recall.*

Low Precision of MUDetect: In AndroidCompass+, the number of misuses and correct API usage was equal. Thus, a random classifier would obtain a precision and recall of 50%. With respect to that, MUDetect tended to perform worse than random, considering their mean precision. We qualitatively checked the patterns used for misuse detection and found no pattern representing the compatibility check, which represents the ground truth API misuse in this dataset. Thus, we assumed that due to non-related patterns, MUDetect was actually like a random classifier, explaining its worse performance in this setting.

Implications We found that

1. RuDetect outperformed MUDetect regarding precision, which, however, required different settings for different training and validation setups (e.g., *within* vs. *cross-project*).

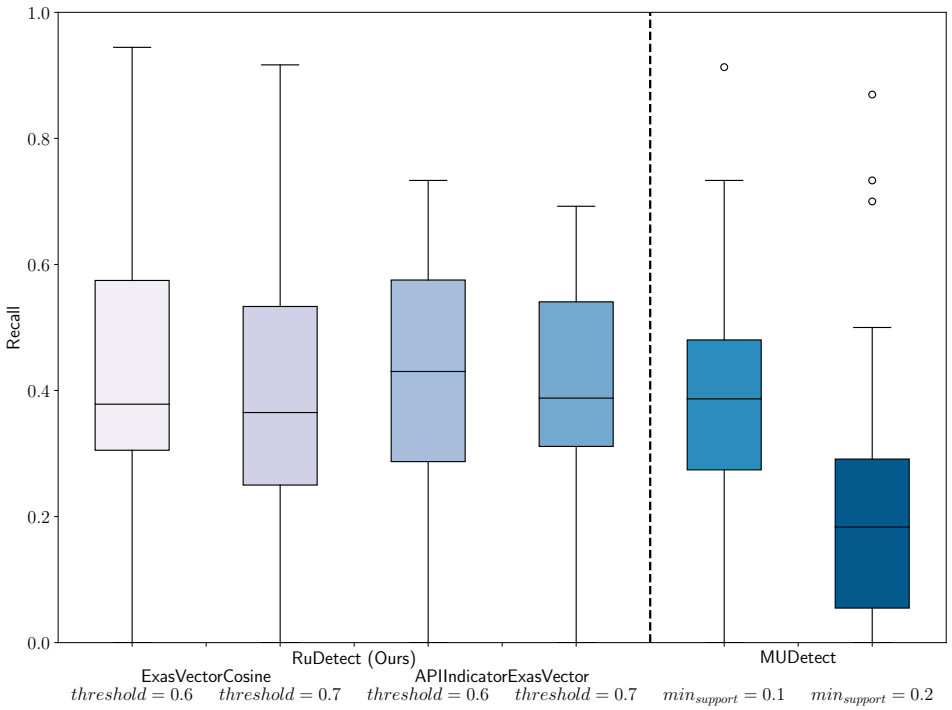


Figure 6.13.: Recall results as boxplot of different settings of RuDetect and MUDetect in the AndroidCompass+ setting

2. RuDetect required a carefully prepared training dataset to obtain large recall, however, the precision was less dependent on the training dataset.

This way, we denote that RuDetect serves as *a precise counterpart to the pattern-based detection* while requiring expert knowledge for configuration towards the aspired use case (e.g., *within* vs. *cross-project*). Moreover, effective application, namely, finding a sufficient number of misuses, requires a good training set containing fixed API misuses that match to the APIs of the tested code. In other words, RuDetect *finds more API misuses (i.e., larger recall) of APIs present in the training set (i.e., fixed certain misuses).*

A major benefit of RuDetect to MUDetect is that it does not require a sufficient frequency (i.e., support) of the change rules. Thus, in its current form, RuDetect *needs a single misuse fix to detect similar misuses*, which simplifies data collection. Moreover, it *avoids the computing effort of FPM* compared to pattern-based misuse detection.

In contrast, *pattern-based misuse detection*, such as MUDetect, *does not require that a misuse is present in the first place*, meaning it may also work if an API has been correctly used before. Moreover, it also supports the case where a fixing change is not present in the VCS, for instance, since a developer fixed the misuse before adding the false version to the repository.

Thus, we conclude that *RuDetect can serve together with pattern-based techniques as a precise complement for API misuse detection.*

6.5.6. Threats to Validity

Once again, our experiments and results are based on empirical methods, which are accompanied by possible threats to validity.

Internal Validity Regarding the internal validity, we validated our misuse detector based on a prototypical implementation based on the AUG implementation of MUDetect[21], as well as the computation of the similarity values as a `Python` implementation. These implementations might contain errors and thus might influence the results we obtained. Even though we thoroughly tested our implementation, we could not guarantee an error-free implementation, neither ours nor from others. For replicability, we provide all experimental data, scripts, and implementation as a replication package[13].

Our validation was based on a ground truth, which could be wrong. Particularly, pre- and post-commit code variants in MUBench [ANN+16] and AndroidCompass+, obtained from its previous version (cf. [NBKO21b]), did not necessarily reflect misuses and correct usages. Similarly, the manual labeling conducted for AU500 [KL21] could contain errors. These issues with the dataset could influence the measurement of precision and recall. Additionally, we corrected the MUBench dataset regarding some entries to link to the correct commit. In case this is done wrongly, our ground truth entries could be wrong as well.

Even though we avoided data leakage, namely, finding misuses by the fix itself, the data still might contain the fix due to forked projects or equal changes in another code location. Moreover, our evaluation did not considered temporal bias. Temporal bias means, in our case, that patterns and change rules were inferred from code, which was not available at the time the misuse was introduced. This way, in practice, the real number of true positive might be lower than reported.

[21] available at `https://github.com/stg-tud/MUDetect` last accessed: 2024/07/16

When mining patterns, MUDetect used all methods from the complete source file of the fixed misuse, while RuDetect only analyzed the method containing the misuse. This way, MUDetect had more training data than our detector. We could not definitely judge whether this had a positive or negative impact. On the one hand, more patterns were mined from other methods, which increased the chance of detecting otherwise non-detected misuses from which the recall and probably the precision could benefit. On the other hand, more patterns might also increase the chance of more false positives and negatively impact precision. Note that in the AndroidCompass+ experiment, we applied MUDetect with relative precision (i.e., using $min_{support}$). Since a larger amount of API usages existed, MUDetect could not find patterns obtaining larger precision and recall due to too-low support. This setting, however, did not apply to the MUBench-on-MUBench and MUBench-on-AU500 settings, in which we used fixed absolute support (i.e., $min_{support_{abs}}$). We found that using the whole source file was a more natural way of applying MUDetect since, usually, during mining, one did not know the correct usage. However, we observed a positive impact on FPM-based misuse detectors using search and filter strategies, as discussed in Chapter 5. This influence on precision and recall was not evaluated, and thus, future research should do so.

Finally, for statistical tests, we applied the Wilcoxon-Mann-Whitney rank sum test. This test only detects *significant differences in the distribution, not necessarily in the mean values*. While we kept this in our minds when judging our results, we strongly emphasize to also consider the results with respect to actual mean differences.

External Validity Regarding external validity, we were limited regarding our applied datasets MUBench [ANN+16], AU500 [KL21], and AndroidCompass+ (obtained from its previous version in our previous work [NBKO21b]). These only represented a subset of APIs and misuses and did not necessarily represent API misuses in the wild. Moreover, we limited our results to the Java programming language and, thus, the object-oriented paradigm.

In our comparison, we only validated against one misuse detector, namely MUDetect [ANN+19b], which limits our discussion regarding improving the state-of-the-art. This experimental restriction was caused by several issues (cf. Section 5.2.2, particularly, **Insight D-2 Insight D-5**). As stated before (cf. **Insight D-1**), we found that simply comparing to the presented results in the paper was not sufficient even though the same dataset was used since the datasets evolved over time or were filtered.

Even though we obtained improvements in the precision, we did compare the usefulness and usability of our approach in a real-world setting. This setting would require additional user studies with developers. Thus, currently, our applicability for practical use cases is only grounded on the concept that change-based misuse detection is constructed to be seamlessly integrated into a VCS.

6.6. Conceptual Differences to Related Work

In this section, we present the conceptual differences between RuDetect and other API misuse detectors (cf. Section 6.6.1), techniques supporting API evolution (cf. Section 6.6.2), and other code change datastructures (cf. Section 6.6.3).

6.6.1. Conceptual Differences to Other API Misuse Detectors

According to the classification of API misuse detectors in Section 5.2.2, we classify RuDetect as *single usage, explicit specification (SES)*. Moreover, we also provide a conceptual comparison of RuDetect to all misuse detectors targeting the Java programming language and applying a replication package (cf. Section 5.2.4) using the characteristics from Table 5.1 in Table 6.15. We observed that RuDetect, compared to ALP [KL21], FuzzyCatch [NVN20], Jadet [WZL07], MUDetect [ANN+19b], and Salento [MCJ17], uses code changes of client code (i.e., C_Δ) as an additional source. CrySL [KSA+21] leverages the knowledge of experts to infer specifications. CL-Detector [ZCSZ21] additionally uses the API code (i.e., C_A), and the technique by Li et al. [LZT+24] uses the API code as well as its documentation (i.e., C_A and D). In contrast, RuDetect does not require knowledge of API code. Similar to RuDetect, CPAM [LCP+21] also uses additional code changes (i.e., C_Δ).

A benefit of RuDetect is that it does not require the knowledge of the misuse API to find related client code (i.e., code collection in a cross-project setting using API-specific information $\times_A$) as of ALP, CL-Detector, Li et al., and Salento. Moreover, RuDetect also infers change rules of internal code (i.e., I) in comparison to Jadet, which solely relies on this setting, and similarly to MUDetect, which we compared in Section 6.5.5. Note that only FuzzyCatch by using error-handling code and CPAM by using change-related code from VCS also uses other information to find cross-project client code (i.e., $\times_O$).

RuDetect only uses Static Code Analysis (SCA), namely, ChaRLI, to infer specifications from the VCS. All other techniques require additional FPM (i.e., ALP, CL-Detector, CPAM, FuzzyCatch, Jadet, MUDetect, Li et al.), a technique from machine learning (i.e., ALP, Salento), or rely on a completely manual process (i.e., CrySL). While CPAM by Liu et al. [LCP+21] also obtains so-called 'change patterns,' their technique does not directly apply these patterns for misuse detection, but rather, the authors manually implement static code analyzers to check these patterns. In contrast, RuDetect can directly apply change rules for misuse detection and, thus, can be easily extended with further change rules.

6.6.2. Conceptual Differences to API Evolution Techniques

In Section 3.4.3, we introduced the background of API evolution techniques. As seen with AndroidCompass+, deprecated APIs can be interpreted as API misuses. Thus, we shortly discuss the main differences of RuDetect to techniques from API evolution.

Most related are techniques for detecting the usage of deprecated API elements (i.e., API elements that are marked as becoming outdated and, thus, potentially causing breaking changes in subsequent API versions). Moreover, API migration techniques to support the transformation of client code to comply with the latest API version (i.e., migrating between API versions) share similarities to our work. However, techniques supporting the detection of deprecated APIs, as well as the migration of API elements, have a special focus on the API evolution and leverage specific characteristics of breaking changes.

API Deprecation Regarding the detection of deprecated APIs, we observe that techniques rely on the API library code. For instance, *Deprecation Watcher*, introduced by Zhou and Walker [ZW16], is based on a set of deprecated API elements obtained from annotations in the documentation and from missing API elements obtained from the code changes of the library. Another technique, named *APIScanner* by Vadlamani et al. [VKC21], leverages

decorators, code warnings, and comments in the library code as indicators for deprecation. In contrast, RuDetect does not require such an analysis of the library's code and thus is not restricted to API misuses based on deprecation.

API Migration On the other hand, API migration techniques between API versions aim to find different kinds of *migration mappings* between API elements, such as one-to-one (e.g., replacement of an API method by another), one-to-many (e.g., replacement of an API method a set of methods), many-to-one (e.g., replacement of many API methods with a single one), or many-to-many (e.g., replacement of a set of API methods by another set) [RBK+13]. While such mappings are similar to change rules, in this form, they lack the information of their dependencies (e.g., data- and control flow represented in AUGs, and thus their change rules). Therefore, some techniques also provide *edit scripts* or migration rules, describing how a specific API usage has to be changed according to the new version [RBK+13]. However, the goal of edit scripts is to update the code instead of detecting potential misuses. Thus, RuDetect can be considered as an extension that leverages migration rules for misuse detection. However, we observe that many migration techniques, similar to API deprecation detection techniques, require the API library code for API mapping inference, for instance [SJM08], *LibSync* by [NNW+10], *AURA* by [WGAK10], *HiMa* by [MWZM12], *Mediator* [XDM19], *RepFinder* by [HCP+21], *A3* by [LSC22], *AUGraft* [WY22], or *MELT* by [RML+23].

Techniques learning migrations from client code are more likely to be found when migrating between programming languages [NNPN17, CXLX21b, ZWX+23] or between libraries [TFB13, DND+25]. Such migrations, however, usually do not target to fix an API misuse and, if so, only with huge effort (i.e., replacing a whole library or even the programming language).

6.6.3. Conceptual Differences to Code Change Datastructures

We applied the AUG as graph-based datastructure to obtain change rules. In the past, many other – also graph-based – datastructures were suggested to represent code changes. A frequently applied tool is GumTree using differences of Abstract Syntax Trees (ASTs) originally using tree-edit distance to obtain edit scripts of ASTs [FMB+14]. In its current version, the authors applied a heuristic to increase efficiency [FM24]. For instance GumTree is used forAPI code recommendations [NHC+16]. However, it only infers structural differences, while AUGs represent control and data flow information.

Similarly, other graph-based structures representing code changes, such as the *Source Code Graph* [NMR22] for bug prediction, a graph structure to infer typical change patterns in VCS [JM22], or the graph datastructure to predict commit messages from code changes [DLZ+22] are focused on structural differences.

In contrast, Nguyen et al. [NND+19] introduced the *fine-grained program dependence graph* (*fgPDG*). This graph reuses elements from the Graph-based Object Usage Model (Groum) and thus represents control and data flow as well. However, it also represents details on different structural elements, such as a `for-each` versus a simple `for` loop. Instead, the AUG by Amann et al. [ANN+19b] is constructed to abstract these features to represent specific API usage with fewer nodes. This way, we expected our change rules to consist of fewer nodes, and thus fewer entries in the respective Exas vectors to better serve for API misuse detection. However, RuDetect is also applicable with other data structures as long as we can compute similarities among them.

Table 6.15.: Conceptual Comparison of RuDetect to other Java-based and available API Misuse Detectors based on the Table 5.1.

	RuDetect (Ours)	ALP [KL21]	CL-Detector [ZCSZ21]	CPAM [LCP+21]	CrySL [KSA+21]	FuzzyCatch [NVN20]	Jadet [WZL07]	MUDetect [ANN+19b]	Li et al. [LZT+24]	Salento [MCJ17]
Type	SES	IS	SES	SES	MES	SES	SES	SES	MES	IS
S	C_C, C_Δ	C_C	C_C, C_A	C_C, C_Δ	E	C_C	C_C	C_C	C_C, C_A, D	C_C
C	I, $\times_O$	$\times_A$	$\times_A$	$\times_O$	-	$\times_O$	I	I, $\times_A$	$\times_A$	$\times_A$
I	SCA	SCA, FPM, AL	SCA, FPM	SCA, FPM	manual	SCA, FPM	SCA, FPM	SCA, FPM	SCA, FPM	SCA, DL
P	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓

Type: Single usage, explicit specification (SES), multiple usage, explicit specification (MES), implicit specification (IS); **Source Data for Specification Inference (S):** client code (C_C), API library code (C_A), changes in client code (C_Δ), API documentation (D); other external sources (E); **Collection Technique of Client Code (C):** not applicable ($-$), code from intra-project setting (I), code from cross-project setting ($\times$) with API-specific information (e.g., class, method, or parameter type names) ($\times_A$) or with other information ($\times_O$) ;**Inference Technique (I):** Static Code Analysis (SCA), Frequent Pattern Mining (FPM), Active Learning (AL), Deep Learning (DL); **Post-processing (P)**

6.7. Summary Change Rule-Based API Misuse Detection

Summary In this chapter, we introduced the concept of *change rule-based API misuse detection* named RuDetect (i.e., **RQ D-R**) as an alternative approach to state-of-the-art *pattern-based API misuse detection* discussed in Chapter 5.

RuDetect is based on *change rules*, representing the essential changes of an API misuse fix obtained via a VCS. Change rules describe the transformation between AUGs (i.e., graph representation of an API usage), namely in the form $aug_{m'} \rightarrow aug_{f'}$. This way, it describes how an AUG $aug_{m'}$ representing a misuse is transformed into an AUG representing its corresponding fix $aug_{f'}$. We introduced an automated technique, named *ChaRLI*, to *infer changes rules automatically* from commits with little manual effort from the developer (i.e., essentially locating the API misuse fix). ChaRLI is integrated into RuDetect. In detail, ChaRLI infers change rules *without* and *with context*, the latter including non-changed code context information (e.g., parameters or surrounding methods).

Having a set of change rules inferred by ChaRLI, RuDetect conducts an *applicability check*. This check assesses whether a change rule is capable of deciding whether an API usage is a misuse or not. We introduced and evaluated two possible variants for the applicability check, namely, a *threshold-based* (i.e., the misuse part of a rule is sufficiently similar to the API usage) and a *control-group-based applicability check* (i.e., the change rule performs sufficiently well in comparison with a ground-truth control group of known misuse and correct API usages). In case a change rule is applicable for the API usage, RuDetect conducts a *graph similarity-based misuse detection* between the AUG of the tested API usage and the applicable rule. Essentially, RuDetect denotes an API usage as misuse if it is more similar to the misuse part of a rule (i.e., $aug_{m'}$) than to fix part (i.e., $aug_{f'}$).

For this purpose, we applied *heuristics to cope with the NP-hard problem of mapping AUGs* [ZTW+09], as well as *tested different variants of graph similarities*. We validated *the best-performing variant* as well as compared it to the state-of-the-art pattern-based misuse detector MUDetect [ANN+19b]. In detail, we evaluated *precision and recall* using different experimental settings based on misuse datasets MUBench [ANN+16], AU500 [KL21], and AndroidCompass+, an extension of our own dataset AndroidCompass [NBKO21b].

Contribution RQ D-R This chapter focused on the impact of applying *change rules* obtained from code changes of previously fixed API misuses for API misuse detection.

In detail, we found that ChaRLI effectively automated the change rule generation based on previous commits (i.e., 78.8% of change rules for MUBench were generated) accompanied by minimal effort from a developer (cf. **Insight D-12** 'Applicable Automated Change Rule Inference' on page 162).

We further introduced the API misuse detector RuDetect and experimentally determined its best setup regarding its precision. We found the best performance when using the *threshold-based applicability check with threshold* = 0.6 and applying the *Exas vector*-based similarity measurement based on the work by Nguyen et al. [NNP+09a] with our extensions to handle API usages (i.e., `Indicator-` or `-Split-` extensions) as shown in **Insight D-13** 'Best setup for RuDetect' on page 173. Moreover, we found that choosing the *threshold* value of the applicability check is crucial to obtain high precision (i.e., **Insight D-14** 'Careful selection of the *threshold* value when using the *threshold*-based applicability check for RuDetect' on page 173).

In addition, we also observed that change rules *with context* achieve higher recall without harming the precision (i.e., **Insight D-15** 'Small but positive effect of change rules

with context on the recall of RuDetect' on page 178) and that the *threshold*-based applicability check is more effective regarding precision than the control-group based variant (i.e., **Insight D-16** 'Positive effect of the *threshold*-based over the control-group-based applicability check for RuDetect' on page 184).

In comparison to the pattern-based API misuse detector MUDetect [ANN+19b], we found that RuDetect performs better regarding the precision depending on the selected similarity measurements (i.e., **Insight D-17** 'Dependency on Similarities and *threshold* for improved Precision of RuDetect compared to MUDetect' on page 187) and comparable regarding the recall, when carefully selecting the training dataset (**Insight D-18** 'Dependency on the training set for a larger recall of RuDetect compared to MUDetect' on page 192). Interestingly, we observed a smaller impact of the training set on the precision than on the recall (i.e., **Insight D-19** 'No strong dependency on the training set for larger precision of RuDetect compared to MUDetect' on page 192). We emphasized that *RuDetect required only a single example of an API misuse fix as a change rule to detect similar kinds of misuses.* Taking the above-mentioned factors regarding RuDetect's performance into account, we found that the change rule-based API misuse detection performed significantly better compared to MUDetect (i.e., **Insight D-20** 'RuDetect significantly performs better than MUDetect' on page 194).

Thus, we answer **RQ D-R** as follows:

RQ D-R What is the impact of applying change rules inferred from previous fixes of API misuses on API misuse detection?

We found that (1) change rules based on AUGs were applicable in a developer-related context, and our technique, ChaRLI, could automatically and effectively infer them. Moreover, we found that applying these rules for misuse detection using our technique RuDetect performed best regarding precision with a (2) threshold-based applicability check with a carefully selected threshold and (3) using variants of Exas vector-based graph similarities for a subsequent graph-based API misuse detection. In addition, we observed an improved recall of RuDetect when (4) enhancing change rules with code context and (5) carefully preparation of the training set. This way, (6) RuDetect performed better than a state-of-the-art pattern-based API misuse detector.

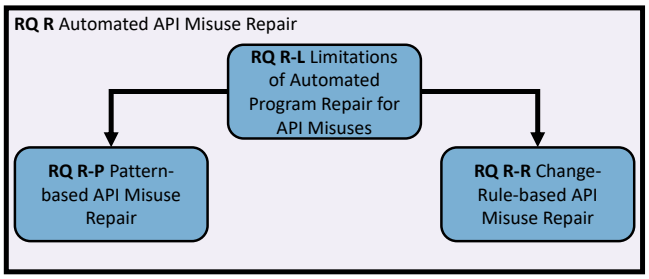
7

Towards API Misuse Repair

This chapter is based on publications from the author, partially together with other colleagues published and presented at the International Conference on Automated Software Engineering (ASE) 2017 in the Doctoral Symposium track [Nie17] at the International Workshop on Automated Program Repair (APR) 2024 [NBKO24].

In this chapter, we analyze the ability of previous Application Programming Interface (API)-specific information from API misuse detection to automatically repair misuses. API misuse repair becomes necessary when the root causes of API misuse cannot be prevented, as discussed in Chapter 4, and when automated techniques for detection, as discussed in Chapters 5 and 6, find these misuses. For repair, we re-use patterns (cf. Chapter 5) and change rules (cf. Chapter 6).

7.1. Methodology and Structure



In this chapter, we target **RQ R** on whether the specific knowledge from API misuse detection supports techniques for Automated Program Repair (APR) to automatically fix API misuses. First, we analyze the limitations of the state-of-the-art APR techniques regarding their ability to target API misuses (i.e., **RQ R-L**). Then, we suggest potential improvements and validate their ability to support APR techniques by applying the previously introduced patterns (i.e., in **RQ R-P**) and change rules (i.e., in **RQ R-R**).

Again, we answer the questions using elaborated scientific methodologies for software engineering research introduced by Ralph et al. [RAB+20][1].

In detail, we answer **RQ R-L** by reviewing the literature on APR research in Section 7.2. Note that due to the large activity in this research field with a number of existing surveys, reviews, and studies [LGFW13, Mon18a, Mon18b, LGPR19, KMSH21, ZFM+23], we do not conduct a separate Systematic Literature Review (SLR) as done in Chapters 4 and 5. We rather use the knowledge from these studies to assess the degree of existing techniques to target API misuses. This knowledge is sufficient since our goal is to assess the *ability*

[1] https://www2.sigsoft.org/EmpiricalStandards/ last accessed: 2025/03/07

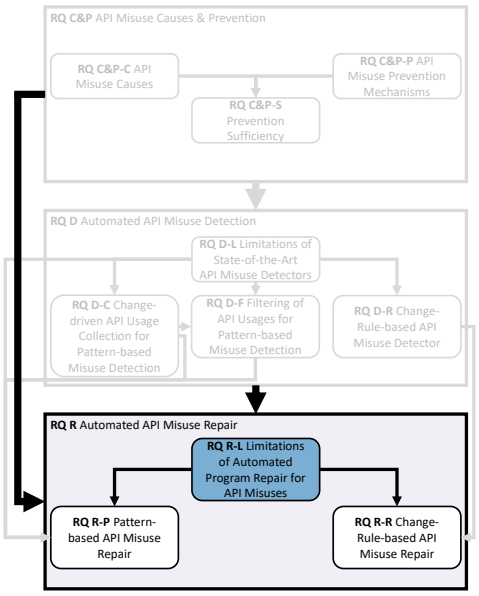
of API-specific knowledge to support APR, while a full-fledged technique for automated repair of API misuse will be part of future work.

Nevertheless, we assess this ability of API-specific knowledge for APR in **RQ R-P** and **RQ R-R** by applying engineering research [RAB+20]. In detail, we suggest a concept and a software artifact named *API-Specific Automated Program Repair (ASAP-Repair)* in Section 7.3 that leverages artifacts from the previously introduced *pattern-based* (cf. Chapter 5) and *change rule-misuse detection* (cf. Chapter 6). We present our empirical evaluation of these techniques (cf. Section 7.4) and their results (cf. Section 7.5).

We conclude in Section 7.6 the current ability of our technique for APR for API misuses as well as the next research steps to accomplish a full-fledged API-specific APR technique.

7.2. Limitations of State-of-the-Art API Misuse Repair

In this section, we address **RQ R-L** by first considering related work on APR (cf. Section 7.2.1) and second, taking a closer look at how existing techniques are limited in automatically repairing API misuse (cf. Section 7.2.2).



7.2.1. State-of-the-Art on Automated Program Repair

Collecting the State-of-the-Art As mentioned before, we did not conduct a separate literature review collection on APR but rather rely on existing studies. For that purpose, we analyzed a set of known articles and references considering different aspects of APR. These articles were found by regularly observing the research progress in APR. In detail, we focused on articles concerned with

- general terms and challenges on APR research [LGFW13, LGPR19];
- a collection of general APR techniques [Mon18a, Mon18b]
- a collection of APR techniques using so-called *learning-based* APR, especially applying machine learning [ZFM+23]
- an overview and challenges of benchmarks for APR research [RRWF25]
- a comparison study of APR techniques for API misuses [KMSH21]

Moreover, we also referred to the community web page `https://program-repair.org/`[2] on automated program repair, including a regularly updated overview of published articles, benchmarks, and software artifacts.

General Terms on and Challenges of APR APR denotes techniques to automatically find and fix faults in software [LGFW13, LGPR19]. According to Le Goues et al. [LGPR19], an APR technique typically consists of the following three steps:

[2] last accessed: 2025/03/07

1. *Bug Detection*, namely, the recognition of a faulty behavior and the location of responsible code snippets;

2. *Patch Generation or Synthesis*, namely, the modification of the software with the goal to overcome the faulty behavior;

3. *Patch Validation*, namely, the process of checking whether the generated patch obtains the goal of overcoming the faulty behavior and does not introduce new faults.

General Challenges: Since the detection and patch validation are undecidable problems, the same is true for general APR [LGPR19, NL22]. Nevertheless, for certain fault types and with specific assumptions, APR is possible [LGPR19].

Le Goues et al. [LGFW13] discussed the main research challenges for APR based on their *scalability*, namely, applying APR techniques to large-scale software products and their *generality*, namely, to which degree APR techniques can be applied to different kinds of software defects. Moreover, Le Goues et al. [LGFW13] emphasized the necessity to reduce and navigate the typically *huge search space of patch templates* to improve the *patch quality* as well as to introduce techniques to *validate generated patches*.

Industrial Application: Despite these limitations, open-source and industrial development applied APR, such as on the Linux kernel [KBK+17], or customized tools like *CLEVER* at Ubisoft Entertainment SA [NHL18], *Getafix* and *SapFix* at Meta Platforms, Inc. [BSPC19, MBC+19], or *Fixie* at Bloomberg L.P. [KWM+21].

APR Taxonomy There exist various different taxonomies on APR techniques.

State vs. Behavior: Monperrus [Mon18a] classified techniques into *state-based* and *behavioral-based*. In detail, *state-based* repair denotes changing the state of a program, such as memory or input data, while *behavioral-based* repair changes the behavior of a program by changing its code.

API-specificity: We [Nie17] extended this classification by an orthogonal dimension on API-specificity, namely, whether a technique leverages API-specific knowledge (i.e., *API-specific repair*) or not (i.e., *generic repair*). We provide an updated classification of these techniques in Figure 7.1 (cf. Section 7.2.2).

Heuristic vs. Constraints vs. Learning: Le Goues et al. [LGPR19] further classified the APR technique Monperrus [Mon18a] denoted as *behavioral-based* repair regarding their applied approaches for generation or synthesis of patches into

- *heuristic-based*, namely, techniques using a heuristic search on a search space of potential code modifications and validating patches produced by them;

- *constraint-based*, namely, techniques that infer a specification describing correct behavior and, based on that, synthesize a correct behavior;

- *learning-based*, namely, techniques learning correct behavior from previous patches and transforming faulty code based on that knowledge.

Patterns: In their survey on learning-based APR, Zhang et al. [ZFM+23] distinguished learning-based, which they focused on machine learning techniques from so-called *pattern-based APR*. The latter they denote as techniques inferring templates of correct usage and applying them for patch generation.

Test-Based Detection and Patch Validation A large number of APR techniques require automated tests for *bug detection* as well as *patch validation* steps [LGPR19].

Test-based Detection: In detail, for *bug detection*, APR techniques apply automated test suites to localize the code snippet, which is likely responsible for the recognized bug [LGPR19].

In this regard, a frequently applied technique is *Spectrum-based fault localization (SBFL)* [SB22, ZFM+23]. Its notion, according to Sarhan and Beszédes [SB22], is the assumption of a set of passing and failing test cases from a test suite. Based on these cases, SBFL determines the *suspiciousness* of certain code lines depending on whether they are executed more frequently with failing than passing tests. A suspiciousness is computed by statistics such as Tarantula [JHS02] or Ochiai [Och57]. Moreover, tools like GZoltar [CRPA12][3] provide a set of different SBFL statistics and are applied by several APR techniques. Despite its frequent usage, SBFL is considered critical for bug localization in large software projects, for instance, since such projects do not have sufficient test cases to cover all program behavior and thus do not correctly localize the root cause of the bug [KGH+17].

Patch Validation: Regarding *patch validation*, a majority of APR techniques validate the *plausibility* of their generated patches by (re-)running the test suite. This way, they assess whether the source code still compiles, the patch passes the previously failed tests, and previously passing tests does not fail [QLAR15, ZFM+23].

However, assuming that many techniques already require a test suite for bug detection, this usually requires an independent test suite. Otherwise, generated patches only satisfy the automated tests used for detection but do not generalize, a problem known as *test overfitting* [SBLGB15, LTLLG18, LGPR19, PMK+24]. A technique to obtain such independent test suites is, for instance, EvoSuite [FA11], which has been applied by Xin and Reiss [XR17] to target overfitting.

Another issue of using test suites for validation is that many potential patch candidates are generated, which require frequent, time-consuming re-compilation and execution of the patched program [ZFM+23]. There exist techniques to speed up the compilation process, such as UniAPR, which applies direct byte-code manipulation in the Java virtual machine [COZ21].

Due to these reasons, in industrial applications, the assessment of whether plausible patches are *valid* is done manually, for instance, in a code review-like process [MBC+19, KWM+21].

Insight R-1 (RQ R-L): Dependency of APR on test suites

Many state-of-the-art APR techniques are dependent on a sufficiently large test suite for bug detection and patch validation, which also causes several issues, such as test overfitting and patch validation efficiency, as well as subsequent efforts to mitigate these effects.

APR Benchmarks It is assumed in APR research to validate their techniques on real-world bugs to accomplish the goal of generality [LGFW13]. Thus, many datasets of different real-world bugs for different kinds of bug types and programming languages were suggested in

[3] latest version available under https://gzoltar.com/ last accessed: 2025/03/06

the past[4]. Prominently used datasets for Java are *Defects4J*[5] [JJE14], *Bugs.jar*[6] [SLL+18], and *Bears*[7] [MUMM19].

These datasets were partially applied in the experiment by Kechagia et al. [KMSH21] in the APIARTy framework[8]. APIARTy contains 101 API misuses from these datasets and different APR techniques for experimental comparison.

Renzullo et al. [RRWF25] discussed the problem that current datasets are typically *too small* to be applicable for machine learning-based APR techniques. Moreover, they discussed issues regarding *data bias* and *data leakage* in these standard datasets. Data bias means that APR techniques overfit to a single dataset, while data leakage denotes that, for instance, a learning-based technique may contain the solution of a bug in their training data and thus just memorize this solution.

Based on this observation, we currently find a lack of sufficiently large API misuse datasets for program repair. On the community web page, we only found the small dataset *Droixbench*[9] with 24 bugs targeting the Android framework. Thus, misuse datasets are smaller in terms of bugs than Defects4J, Bugs.jar, and Bears (i.e., between 251 and 1,158).

Insight R-2 (RQ R-L): Few and too small APR benchmarks for API misuses

State-of-the-art APR research has few (i.e., two datasets) and typically too small datasets (i.e., 24 up to 101 bugs) of API misuses.

7.2.2. Limitations of API-Specific Automated Program Repair

Selection of State-of-the-Art APR Techniques We updated our classification from our previous work [Nie17] based on recent publications. Our focus was to find further APR techniques targeting API misuses. The final classification is depicted in Figure 7.1.

In detail, we used the survey by Monperrus [Mon18a] as well as the frequently updated living review [Mon18b] (i.e., version 6 with timestamp 12-09-2023). Moreover, we considered all APR techniques discussed in the comparative study by Kechagia et al. [KMSH21] as well as a survey on learning-based APR techniques by Zhang et al. [ZFM+23]. Finally, we consulted the publications on the APR community page[10]. Particularly, we reviewed articles having the keyword `API` in their title or description.

In detail, we found by the living review by Monperrus [Mon18b] CDRep [MLLD16] and APIFix [GRS+21] as *API-specific, behavioral APR techniques*. From the community page, we classified TADAF [BOST22] as this APR class.

Limitations State-of-the-Art APR Techniques We present the limitations of the current APR technique shown in Figure 7.1 to effectively repair API misuses.

Generic, State-Based APR: First, we discuss the group of *generic, state-based APR* (cf. bottom, left quadrant in Figure 7.1). The techniques in this group attempt to overcome all kinds of bugs at runtime by changing the state of the program. This state change

[4] cf. community web page `https://program-repair.org/benchmarks.html` last accessed: 2025/03/07
[5] `https://github.com/rjust/defects4j` last accessed: 2025/03/06
[6] `https://github.com/bugs-dot-jar/bugs-dot-jar` last accessed: 2025/03/06
[7] `https://bears-bugs.github.io/bears-benchmark/` last accessed: 2025/03/06
[8] In detail, they used MUBench [ANN+16], which is partially based on *Defects4J*, and bugs from *Bugs.jar* and *Bears*
[9] `https://droix2017.github.io/` last accessed: 2025/03/07
[10] `https://program-repair.org/bibliography.html` last accessed: 2025/02/28

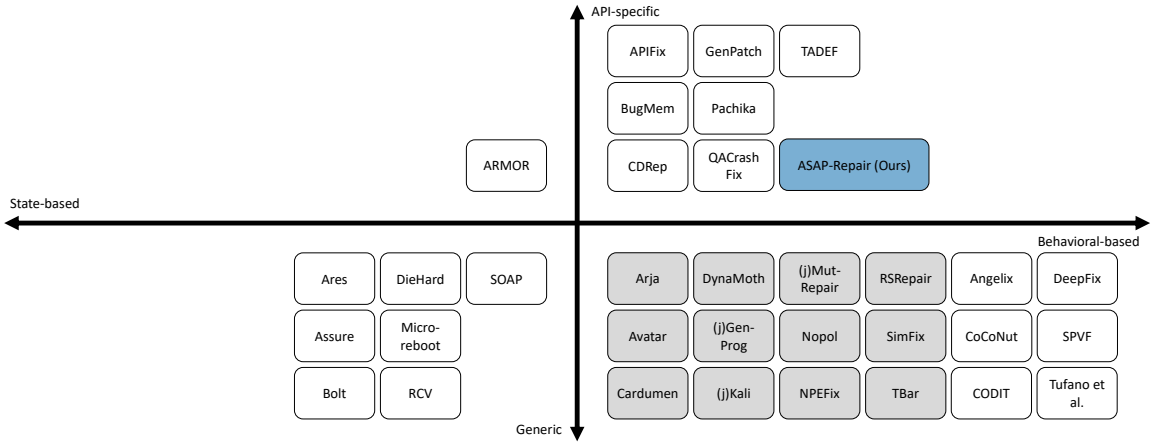


Figure 7.1.: Our updated classification of APR from previous work [Nie17] inspired by Monperrus [Mon18a]. Gray techniques are validated by [KMSH21]. Our technique ASAP-Repair is shown in blue.

may be the execution point (i.e., program counter), for instance, *Assure* [SLP+09] and *Ares* [GSM+16] jump to present exception handling code, or *Bolt* [KMCR12] has strategies to escape from infinite or long-running loops. *DieHard* [BZ06] provides redundancy in heap space to target memory-related bugs. Another strategy is dynamically changing the value of variables and parameters. For instance, *SOAP* [LGC+12] alters the input of a program in case of a crash, or *RCV* [LSDR14] replaces divide-by-zero or null accesses with default values. *Mircoreboot* [CKF+04] introduces a procedure to reboot single components of software to avoid long-lasting system-wide reboots.

These techniques aim to overcome bugs during runtime, for instance, to avoid or mitigate the downtime of server applications, namely serving a *liveness criteria*. This way, these techniques provide temporal support. Moreover, they rely on typically simple and static repair ideas, which do not cover the variance of API misuses. For instance, replacing computation errors with default values may prevent the application from crashing but can hide erroneous behavior (e.g., default values for random variables in cryptography causing security issues). Thus, we conjecture that such techniques are only applicable for API misuses if liveness is an important requirement and if the state change does not hide another misbehavior of the software. On the other hand, these techniques help in case only binaries of the software are available since they do not require access to the source code.

API-Specific, State-Based: Second, we consider API-specific, state-based APR (cf. upper, left quadrant in Figure 7.1). Similarly to generic, state-based APR, these techniques aim to change the state of the software, but they leverage API-specific information. We only found *ARMOR* [CGM+13] as a representative. This technique leverages the redundancy in program libraries, namely API methods with similar behavior. In case of an error in one method call, ARMOR changes the execution trace by calling its respective similar method.

This technique, however, requires the presence of sufficient redundant functions, which is usually considered as an anti-pattern [FBB+14, p. 78]. Moreover, even though the behavior is similar, it is not equal and thus may introduce unexpected changes. Thus, replacing calls should be done with care and not silently to prevent unforeseen behavior.

Insight R-3 (RQ R-L): Limitations of state-based APR techniques as a temporal solution with possible side-effects for API misuses

In general, we conjecture that state-based APR techniques have only limited value in fixing API misuses since they mainly represent a means to temporarily overcome bugs to satisfy the liveness criteria of software and typically provide simple and static solutions, which may introduce unintended side effects.

Generic, Behavioral-Based APR: Third, we consider techniques classified as *generic, behavioral-based APR* (cf. lower, right quadrant in Figure 7.1). These techniques aim to permanently change the behavior of the software by directly changing its code. We consider more traditional techniques discussed by Kechagia et al. [KMSH21] and learning-based techniques surveyed by Zhang et al. [ZFM+23].

Traditional APR: In the classification, we only depict those APR techniques validated by Kechagia et al. [KMSH21]. In their experiment, they analyzed how effectively current generic, behavioral-based APR techniques can fix API misuses. Note that they restrict the APR techniques to target the Java programming language since the dataset in their APIARTy framework is based on API misuses written in Java. Due to this restriction, they applied the respective Java implementation provided in ASTOR [MM16a] for DynaMoth [DM16], for GenProg [LGDVFW12], for Kali [QLAR15], and for MutRepair [DW10] (i.e., prefixed with a '*j*'). Moreover, they disregard learning-based APR techniques since they require further external data, such as other code samples.

A classical approach is the *generate-and-validate mechanism*, meaning the APR technique generates a set of possible patches as modifications of the code, and then check the patches' validity by automated tests [QLAR15, LGPR19].

The variants of patch generation range from

- removing functionality (e.g., Kali [QLAR15]);
- randomly modifying either within a genetic algorithm framework (e.g., GenProg [LGDVFW12], Arja [YB20]) or using full randomness (e.g., RSRepair [QML+14]);
- based on previously determined modification patterns (e.g., MutRepair [DW10], NPEFix [DCSM17], TBar [LKKB19a]);
- or learned patterns from external sources (e.g., Cardumen [MM18], Avatar [LKKB19b], SimFix [JXZ+18]).

Another strategy for patch generation is to use a constraint-based technique as introduced in the *Angelix* APR technique [MYR16]. This technique denotes the analysis of the execution traces from failing tests of a program and replacing certain statements by so-called *angelic constraints or variables* that would change the behavior of the program (e.g., changing the `if`-condition to an angelic condition returning `true` so that the `then`-clause is chosen instead of the `else`-clause). In case a passing execution trace is found, a synthesis mechanism such as Satisfiability Modulo Theories (SMT) is applied to find a matching surrogate statement constructed from a set of possible statements. In our set, DynaMoth [DM16] and Nopol [XMD+17] apply this mechanism.

In their evaluation, Kechagia et al. [KMSH21] found that all existing techniques generate patches for 28% of all tested misuses (i.e., they only tested 89 from those initially selected 101 misuses). From all generated patches – also multiple patches for single misuses – 65% were considered valid. The authors also found that Nopol, Kali, and Arja produced the

most patches (i.e., between $7.8 - 11.2\%$ of all tested 89 misuses), while Avatar and TBar produced the largest proportion of valid patches (i.e., $> 60\%$). Moreover, they found an overlap of generated patches. This overlap was also represented by the low number of valid patches, namely, only nine (i.e., 10.1% from all tested 89 misuses) patches were considered valid (cf. number of "semantically correct" patches in Table 8 in [KMSH21]).

Kechagia et al. also analyzed reasons why APR techniques fail and found that 51% failed due to false fault localization, meaning the techniques fail to identify the location that should be altered in order to overcome the bug. Regarding the kind of API misuses fixed by valid patches, Kechagia et al. identified that mainly missing conditions, expressions, null checks, and values could be targeted.

Insight R-4 (RQ R-L): Non-sufficient solution for API-specific APR by traditional, generic, behavioral-based APR techniques

We determine that traditional, generic, behavioral-based APR techniques do not sufficiently cover API misuses as undermined by the experiments by Kechagia et al. [KMSH21], who observed a lack in the kind of misuses that could be target (i.e., only missing API elements) and issues in effectively localizing API misuses.

Learning-Based APR: Moreover, Zhang et al. [ZFM+23] surveyed various techniques using *learning-based APR*, particularly those applying Deep Learning (DL). In detail, they found that many techniques interpreted the APR problem as a *neural machine translation* task by 'translating' the buggy code into a respective fixed one. Thus, these techniques typically use an encoder-decoder architecture. The general behavior of these techniques is similar to the ordinary APR steps with fault localization, patch generation, and patch validation. The patch generation is proceeded by a learning task, which leverages code samples of bug-fix pairs (e.g., [TWB+19]), edits of previous fixes (e.g., CODIT [CDAR22]) as well as other specification kinds, such as natural language error descriptions (e.g., SPVF [ZBW+22]). The trained model then uses a previously localized buggy code line, partially together with other information like its context, for instance, surrounding lines of code (e.g., CoCoNut [LPP+20]), and generates a set of patches. Subsequently, these patches are ranked and validated. Zhang et al. [ZFM+23] observed that the last step was conducted by executing test suites or measuring patch similarity to a ground truth fix.

While achieving promising results, a downside is the necessity of sufficient, high-quality training data as well as the computational effort of the training [ZFM+23]. Moreover, learning-based APR mostly relies on traditional test-based techniques for fault localization and patch validation, even though some tools such as DeepFix [GPKS17], integrate both steps into an end-to-end machine learning model. Thus, a majority of techniques can suffer from similar issues as observed in traditional APR in test-based detection.

Further, Zhang et al. [ZFM+23] discussed the potential of applying pre-trained models with subsequent fine-tuning to target specific kinds of bugs. For instance, they discussed techniques tailored for security vulnerabilities (e.g., SPVF [ZBW+22]). Nevertheless, Zhang et al. suggested expanding these strategies to other bugs, such as API misuses. Moreover, even though Zhang et al. discussed promising results of techniques using pre-trained models, the question remained whether these models were trained on their actual fixes of the tested datasets, an issue known as *data leakage* [RRWF25]. For instance, the code-aware model CodeBERT [FGT+20] denotes that source code is used from non-forked open-source projects from GitHub and thus may contain the repositories of APR benchmarks such as

Defects4J [JJE14]. We checked the pre-training data from CodeBERT[11] and found that it partially overlapped with projects used in the Defects4Jv2.0 dataset[12]. Thus, we conjecture that this can bias the obtained results.

Insight R-5 (RQ R-L): Promising but still limited applicability of learning-based, generic, behavioral-based APR techniques for API misuses

We determine that learning-based APR included in the category of generic, behavioral-based APR techniques are still limited to target API misuses in spite of reported promising results since they (1) require a large effort for obtaining data and conducting the training, (2) frequently apply test suites for fault localization and patch validation with previously discussed downsides, (3) can suffer from data leakage, and (4) require further effort, such as fine-tuning to target API-specific bugs.

API-Specific, Behavioral-Based APR: Fourth, we discuss *API-specific, behavioral-based APR* techniques (cf. upper, right quadrant in Figure 7.1). These techniques target API misuses either directly, through their repair mechanism, or their data source.

GenPatch [Wei06] leverages state machines representing specifications to detect execution paths that contradict them (i.e., not accepted by the state machines). It patches the code with an increasing number of change operations (i.e., add, delete, or update on statements). In his paper, however, Weimer [Wei06] did not evaluate this approach.

BugMem [KPW06] targets project-specific bugs, particularly by leveraging information from the project's Version Control System (VCS). In detail, they define bug memory as buggy and related code blocks and match similar blocks to these memories. These blocks serve as patch suggestions and thus transfer previous fixes with customized API usages to future scenarios. However, it is limited to the project's history.

Pachika [DZM09] obtains object behavior by inferring a model from test execution. Particularly, they extract preconditions from passing runs and transfer them to code snippets that are correlated with failing tests. Its downside is that it requires sufficient test cases as well as a larger compute time to build and run the tests.

QACrashFix [GZW+15] obtains patch candidates from related Q&A posts, extracts the suggested fixes, and matches them to the buggy code. Thus, it can target API misuses as well. However, they require an explicit crash or error message of the misuse and may suffer from low-quality fix suggestions from Q&A posts [ZUR+18].

Other techniques specialized in specific APIs, for instance, *CDRep* [MLLD16] focuses on cryptographic libraries, and *TADEF* [BOST22] targets TensorFlow. Both apply manually obtained fix patterns tailored to the API domain and thus are limited this way.

APIFix [GRS+21] targets *breaking changes* by inferring so-called *transformation rules* either from the updated test code of the library itself, from client applications that were already updated, or from client code that applies the updated API version. These transformation rules are equipped with *guard* conditions preventing them from being applied on API usages, which do not represent the breaking API. While it does not require tests for fault localization, it is limited to API misuses caused by changes in the library itself (i.e., breaking changes) and requires access to the repository and the source code of the library.

[11] available under https://drive.google.com/uc?id=1xgSR34XO8xXZg4cZScDYj2eGerBE9iGo according to https://github.com/microsoft/CodeBERT/tree/master/CodeBERT/codesearch both last accessed: 2025/03/12

[12] cf. https://github.com/rjust/defects4j/tree/v2.0.0 last accessed: 2025/03/12

Insight R-6 (RQ R-L): Too narrow applicability of API-specific, behavioral APR for API misuses

While we found techniques targeting API misuse in particular, we conjecture that these are limited since they (1) target only specific API misuses (e.g., intraproject, specific domain), (2) require high-quality data (e.g., valid fixes from Q&A pages), or (3) require computational expensive fault localization (e.g., test execution).

7.3. ASAP-Repair: API-Specific Automated Program Repair

We targeted these limitations with our own technique, ASAP-Repair. ASAP-Repair is not a full-fledged APR, and thus, we do not compare it with state-of-the-art APR on an experimental level. However, we give a conceptual comparison to them in Section 7.5.2. For orientation, we classified ASAP-Repair as API-specific, behavioral-based APR and, depending on the classification granularity, as learning-based or patch-based APR.

7.3.1. General Steps of ASAP-Repair

Idea ASAP-Repair reuses the specification by which an API misuse has been detected. Recall the two possible detection techniques, namely, *pattern-based* and *rule-based* detection, from Chapters 5 and 6. By using these two techniques, we do not require tests for fault localization and thus target the issue on the dependency of test suites (i.e., **Insight R-1**) and partially the limitations of generic and API-specific behavioral-based as well as learning-based APR (i.e., **Insight R-4**, **Insight R-5**, and **Insight R-6**).

Thus, ASAP-Repair generates patches with two repair techniques using either *patterns* or *change rules*. We refer to them as *pattern-based* and *rule-based repair*, depending on the applied misuse detection and data structure for repair. Patterns are obtained via *Frequent Pattern Mining (FPM)*, while change rules are provided by previous API misuse fixes marked manually by developers and inferred with our technique *Change Rule Inference (ChaRLI)* (cf. Section 6.3.2). Especially, rule-based repair has significantly less effort for training compared to learning-based APR using DL (i.e., issue in **Insight R-5**).

In our variant, we apply the fix to the intermediate representation of *API Usage Graphs (AUGs)* (cf. Section 3.2.2). This procedure means we edit the misuse AUG using the respective pattern AUG or the change rule used for detection and produce a *fixed AUG*. This way, we provide a permanent solution compared to *state-based APR techniques* (i.e., targeting the issue in **Insight R-3**). The edit denotes addition, deletion, or updates of API elements, and due to the generic structure of AUGs, ASAP-Repair has no restrictions on the misuse type or the kind of fix targeting **Insight R-4** and **Insight R-6**.

Recall that applying the fix to the AUG representation *limits the practical application* while *allowing a more focused scientific analysis* of the ability of ASAP-Repair. We further discuss these limitations and steps towards a practical API-specific APR technique in Section 7.5.3.

Valid Fixes We denote a *fixed AUG as valid* if it produces a *syntactically correct AUG*, which *matches a ground truth AUG* or is *semantically equal* to it. Currently, for *syntactical correctness*, we assume that the fixed AUG *is still a directed acyclic graph*. Note that this can be further extended by constraints such as restricting certain edge types between

particular node types. For *validity*, we compare the fixed AUGs with the ground truth AUGs of fixes given by the original developers. In a practical scenario, this may require the execution of regression tests as well as code review processes to validate automatically generated patches similar to industrial applications of APR techniques [NHL18, BSPC19, MBC+19, KWM+21].

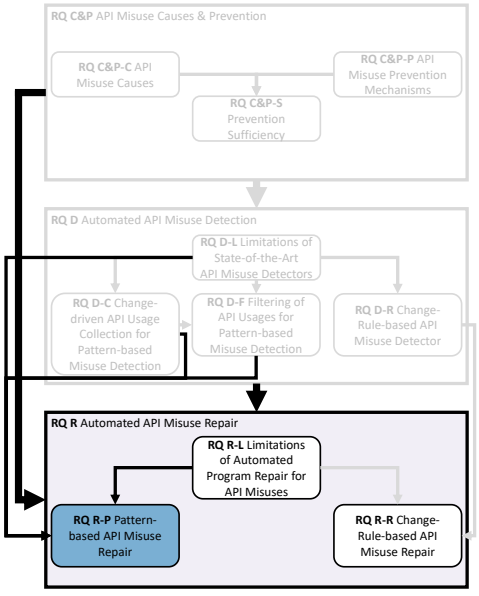
Repair Steps Structure We depict the general concept of ASAP-Repair in Figure 7.2 in a UML-like activity diagram. We refer to this figure by the letters (A)-(H), subsequently. In the following, we briefly recall the misuse detection for ASAP-Repair in the subsequent Section 7.3.2 and discuss the *unique steps of each repair variant* in Sections 7.3.3 and 7.3.4, and the final *transformation step*, namely, how exactly an AUG is transformed in Section 7.3.5.

7.3.2. Misuse Detection in ASAP-Repair

First, we start by transforming a *potential API misuse* into its respective AUG (cf. (A)). After generation, the next step, which is similar to other APR techniques, is the detection of API misuse, namely the recognition and the location. In detail, we apply the already presented techniques of *pattern-based* (cf. (B)) and *change rule-based* (cf. (C)) misuse detection discussed in Chapters 5 and 6. Note that this also requires the inference of patterns by using techniques such as FPM and the inference of change rules using our suggested technique, ChaRLI (cf. Section 6.3.2). Moreover, the applied misuse detection technique defines on the applied repair technique, namely, pattern-based or rule-based.

7.3.3. Pattern-Based Steps of ASAP-Repair

By using pattern-based repair, we target **RQ R-P**. In case a pattern finds a violation within an AUG, this means a certain sub-graph of the misuse AUG contradicts the pattern AUG. One way to fix the misuse is to adapt exactly that sub-graph based on information from the patterns. Thus, the latent assumption is that the misuse is only caused by this local sub-graph part and that no further transformations in other parts of the misuse AUG are necessary to produce a valid fix. This assumption is necessary since the pattern does not provide sufficient information on how the surroundings of the patterns may look like. Furthermore, we require that the misuse AUG is equal or larger in terms of the number of nodes than the pattern AUG since, otherwise, the pattern is not fully covered by the API usage.



Using these assumptions, we first have to identify the relevant part of the misuse AUG to be transformed. We do this by identifying the *external* APIs, typically class names with their related package prefix (e.g., `java.util.List`), used in the pattern (i.e., `getAPIs`-method (cf. (D)). We use these APIs as filters to identify those nodes from the

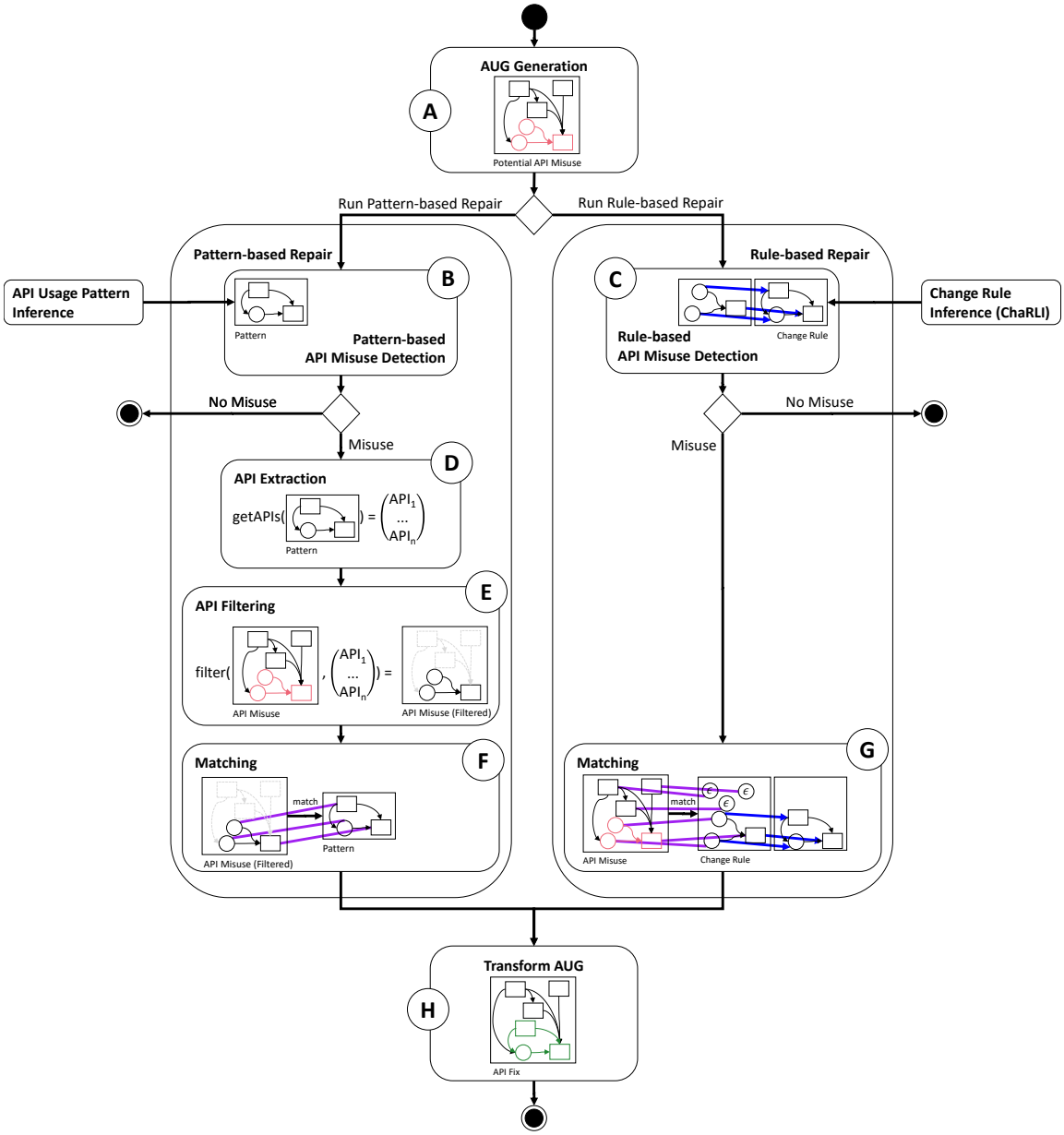
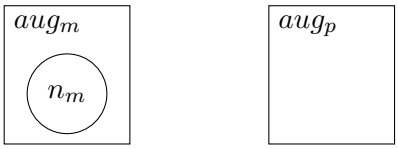


Figure 7.2.: Concept of AUG-based ASAP-Repair including *pattern-based* and *rule-based* repair

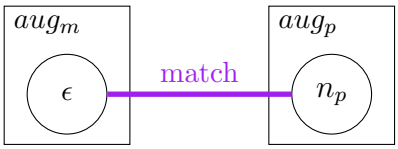
misuse AUG *that should not be transformed*, particularly not deleted when transforming the graph (cf. Ⓔ). Thus, every node that does not relate to the APIs used in the pattern will be kept in the original graph. In the next step, we match the pattern AUG with the relevant sub-graph of the misuse AUG (cf. Ⓕ). We apply the same matching heuristic as discussed in the inference of API change rules in Section 6.3.2 since graph matching or the subgraph isomorphism problem is known to be NP-complete [Epp99, AARRM15]. In detail, we apply the Kuhn-Munkres algorithm [Mun57] using the same customized cost function as used in the change-rule inference to find the minimal cost matching between the nodes from the pattern AUG and nodes from the sub-graph of the misuse AUG. Recall that we mark additions or deletions by special ϵ nodes. In detail, we obtain the following four cases based on the matching between the misuse AUG (i.e., aug_m) and the pattern AUG (i.e., aug_p):

1. *Keep*:



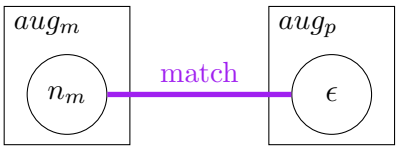
   A node (i.e., non-ϵ) n_m of the aug_m is not matched to any node in aug_p: Node n_m is kept in aug_m since they do not relate to the pattern.

2. *Add*:



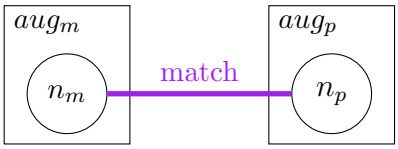
   An ϵ node of aug_m is matched to node n_p (i.e., non-ϵ) of aug_p: This describes the addition of node n_p from aug_p to aug_m together with its respective edges.

3. *Remove*:



   A node n_m (i.e., non-ϵ) of aug_m is matched to an ϵ node of aug_p: This describes the removal of the node n_m from the aug_m together with the respective edges.

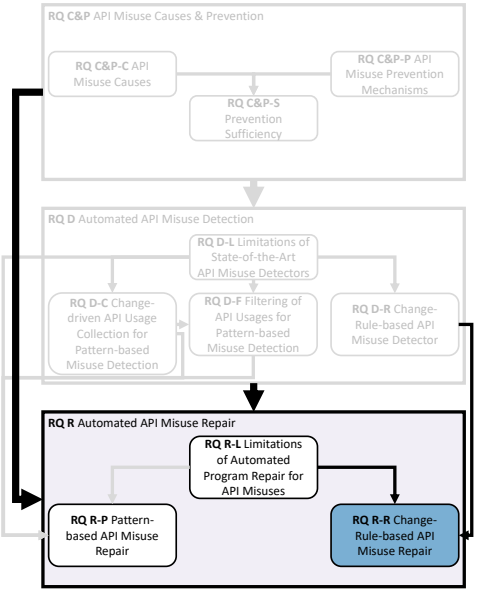
4. *Update*:



   A node n_m from aug_m is matched to a node n_p in aug_p (i.e., both non-ϵ): This describes the update of the node n_m from aug_m to n_p of aug_p together with its respective edges.

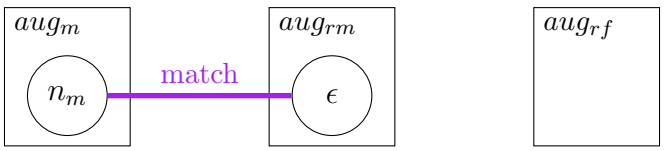
Based on this matching, we transform the respective misuse AUG (cf. Section 7.3.5).

7.3.4. Change Rule-Based Steps of ASAP-Repair



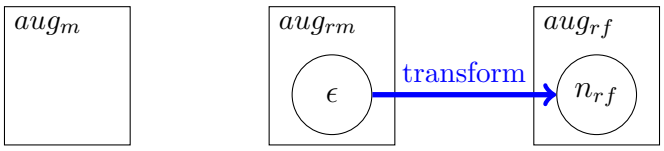
In the second change rule-based repair, we assume that a change rule detected an AUG as misuse targeting **RQ R-R**. Recall that this means that the misuse part AUG of the change rule is more similar to the misuse AUG than to its fix part AUG (cf. Equation 6.3 in Section 6.3.4). Similar to the pattern-based repair step, we have to identify the part of the misuse AUG that has to be changed. However, as described in the following, this is much simpler. In detail, we match the nodes of the misuse part AUG of the change rule with the nodes of the misuse AUG (cf. Ⓖ). Note that we assume that the misuse AUG is equal or larger than the misuse part AUG (i.e., in terms of number of nodes) and that the misuse can be fixed solely on the information within the change rule. In contrast to the pattern-based repair, we match the complete misuse AUG. Since we expect that the misuse AUG is typically larger than the misuse part AUG, some nodes of the misuse AUG are matched to ϵ-nodes. These nodes will be kept in the graph since they are not part of the described change. All other nodes are mapped to nodes present in the misuse part AUG and thus will be transformed. When combining this matching with the complete change rule as depicted in Figure 7.2, we obtain a triple matching between the nodes of the misuse AUG (i.e., aug_m), the misuse part AUG (i.e., aug_{rm}), and the fix part AUG (i.e., aug_{rf}). Based on this triple matching, we can derive the following four transformation cases:

1. *Keep*:



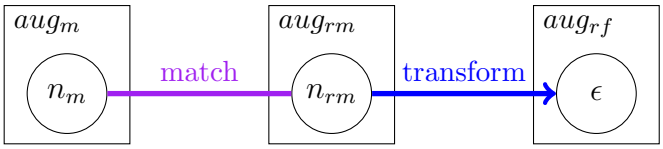
A node n_m (i.e., non-ϵ) of aug_m is mapped to an ϵ node in aug_{rm}: The node n_m remains in the aug_m since it cannot be matched to the misuse situation described in the change rule.

2. *Add*:



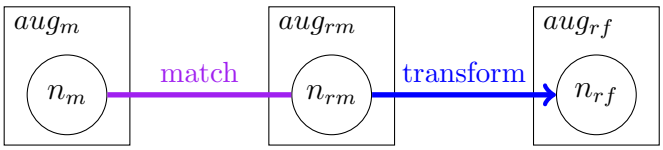
An ϵ node of aug_{rm}, which is not matched to any node of aug_m, is matched to a node n_{rf} (i.e., non-ϵ) of aug_{rf}: This describes the addition of node n_{rf} from aug_{rf} to the aug_m together with the respective edges.

3. *Remove*:



A node n_m (i.e., non-ϵ) of aug_m is matched to a node n_{rm} (i.e., non-ϵ) of aug_{rm}, which itself is matched to an ϵ node of aug_{rf}: This describes the deletion of node n_m in aug_m together with its respective edges.

4. *Update*:



A node n_m of aug_m is matched to a node n_{rm} of aug_{rm}, which itself is matched to a node n_{rf} of aug_{rf} (i.e., all three non-ϵ): This describes the update of the node n_m in aug_m to the node n_{rf} in aug_{rf} together with its respective edges.

The exact transformation is described in the following section.

7.3.5. AUG Transformation of ASAP-Repair

The final repair step is the transformation of the misuse AUG based on the previously derived transformation steps (cf. (H)). As discussed before, both repair variants (i.e., pattern- and rule-based) have four cases of AUG node transformations, namely *keep*, *add*, *remove*, and *update*. Thus, the respective transformations in the AUG are therefore similar for both repair variants. We split the transformation into two phases: in the first one, we *handle the node transformations*, and in the second one, *we do the edge transformation of the AUG*.

According to the first phase, we *add*, *remove*, and *update* nodes, where updating means relabeling nodes if necessary. Since these node transformations in phase one are independent of each other, the order of their execution is independent as well.

Regarding phase two, we first *remove* all edges connected to nodes removed in the previous step and *add* all edges introduced by added nodes (i.e., which are present in the pattern or in the fix part AUG of the change rule, respectively). Note that by construction, all respective counterpart nodes of the added edges will be present in this phase. Finally, we update all edges of updated (i.e., relabeled) nodes. For that purpose, we first remove all incoming edges of the updated node present in the misuse AUG. Note that we only have to remove the incoming edges since outgoing edges, which have to be updated, will be handled when the respective target node is transformed. This transformation *will* happen as this node requires either a *remove* or an *update* operation due to the update of the edge. Then, we add all edges present in the pattern AUG or fix part AUG of the change rule. Using this approach, we safely remove all unnecessary edges while adding all edges required for the patch.

7.4. Experimental Data and Processing

In this section, we describe our datasets and the experimental settings of ASAP-Repair.

7.4.1. API Misuse Datasets

MUBench Again, we applied the MUBench dataset [ANN+16] using the 116 entries as discussed in Section 6.4.1 for an idealized check of ASAP-Repair. Note that we mentioned in that section that three entries were not available anymore from their respective repositories. These entries will be highlighted in the respective results. For all entries, we consider the pre-fixed version as misuse and the post-fixed version as correct API usage.

AndroidCompass+ We also used the AndroidCompass+ dataset as presented in Section 6.4.1. It consists of 36 clusters of API misuses clustered by methods from the Android framework protected by a compatibility check in an `if`-condition. In sum, these 36 clusters encompass 1,018 different method declarations, and thus, 1,018 different AUGs, containing 1,317 different protected methods, and thus 1,317 different single API misuses. Similar to MUBench, this dataset contains the pre- and post-fixed, which we consider as misuse and correct usage, respectively. This way, we obtain a larger dataset of misuses targeting the issue of too small API misuse datasets (cf. **Insight R-2**).

Excluded Datasets We do not use the AU500 dataset [KL21] in this setting because we observed non-promising results on misuse detection when inferring patterns and change rules from MUBench and applying it on AU500 (cf. Section 6.5.5).

Additionally, we do not apply the dataset *APIRepBench* applied by Kechagia et al. [KMSH21] consisting of 101 misuses and obtained from MUBench (i.e., 42 entries) as well as the datasets *Bears* [MUMM19] (i.e., 19 entries) and *Bugs.jar* [SLL+18] (i.e., 40 entries). The reasons are that (1) we already use a significant part of APIRepBench by MUBench, and (2) we require some training data, namely correct API usages and previous fixes to infer change rules, which are not available for APIRepBench.

7.4.2. Comparing Patched AUGs with Ground Truth AUGs

To assess the quality of a generated patch previous APR techniques use automated tests to check their validity (i.e., passed tests) [LGPR19], which was also applied in the APIARTy framework [KMSH21]. However, test case-based validation does not apply to the current form of ASAP-Repair, which *does not produce patched source code but patched AUGs*.

Therefore, we require a technique to compare a patched AUG with its ground truth AUG, namely the AUG generated from the manually fixed source code. We do this comparison by finding the minimal matching between the two AUGs. As previously discussed, minimal graph matching is known to be an NP-complete problem [AARRM15].

Thus, we applied a similar heuristic already presented for the inference of change rules using ChaRLI (cf. Section 6.3.2, namely, in paragraph *Efficient Change Rule Inference*). Recall that we construct a bipartite graph with each partition consisting of the nodes of the two respective AUGs and connecting all nodes by edges labeled with their respective *cost*s representing the number of necessary edits (i.e., adding, deleting, updating the node and its incoming and outgoing edges). In contrast to ChaRLI, we applied for each edit the $cost = 1$ since we were only interested in the number of different nodes and edges. Then, we

applied the Kuhn-Munkres algorithm [Mun57] to find a one-to-one matching with minimal costs, as well as to determine and mark the differences between the two AUGs.

Based on this matching, we compute the *node* (i.e., sim_{node}) and *edge similarity* (i.e., sim_{edge}) of the two AUGs to represent the similarity of the generated patch AUG aug_i and its ground truth aug_j. Formally[13], we assume a matched AUG:

$$aug_{matched} := (V_m, E_m, \Sigma_{V_m}, \Sigma_{E_m}, s_m, t_m, l_{V_m}, l_{E_m})$$

between two AUGs aug_i and aug_j, with

- V_m and E_m representing the nodes and edges with their related costs w.l.o.g. to transform aug_i to aug_j, including those nodes which would be removed;
- their related alphabets Σ_{V_m} and Σ_{E_m};
- their mapping functions of edges to start (i.e., s_m) and target (i.e., t_m) nodes; and
- their labeling functions l_{V_m} and l_{E_m}

Further, we define the minimal costs as obtained by the minimal matching between the nodes and edges of aug_i and aug_j via the Kuhn-Munkres algorithm as functions:

- $cost_{KM_{node}} : V_m \to \mathbb{N}_0$ and
- $cost_{KM_{edge}} : E_m \to \mathbb{N}_0$.

Based on the matching AUG $aug_{matched}$ and the respective cost functions $cost_{KM_{node}}$ and $cost_{KM_{edge}}$, we compute the similarity of the nodes and edges as follows:

$$sim_{node}(aug_{matched}) = \frac{|\{n | n \in V_m \wedge cost_{KM_{node}}(n) = 0\}|}{|V_m|}$$

$$sim_{edge}(aug_{matched}) = \frac{|\{e | e \in E_m \wedge cost_{KM_{edge}}(n) = 0\}|}{|E_m|}$$

Essentially, these functions compute the ratio of shared nodes and edges that do not require edits between the two AUGs aug_i and aug_j. We automated this comparison as an AUG comparison technique.

7.4.3. Experimental Settings

Idealized Check with MUBench We used the MUBench dataset [ANN+16] to assess the repair ability of ASAP-Repair. For that purpose, we compared the pattern-based and the change rule-based variants by conducting *an idealized check*. This setting meant we repair the misuse entries of MUBench with their own fix.

For *pattern-based repair*, we applied the AUG generated from the fixed method declaration of the respective misuse as a pattern. Regarding the *change rule-based repair*, we inferred the change rule using the misuse itself and its fixed version.

This way, we assessed whether ASAP-Repair *was able to produce patches or not* and whether the produced *patches were valid*. In detail, we denoted a patch to be valid if $sim_{node} = 1 \wedge sim_{edge} = 1$ or if our *manual assessment* denoted that the differences to

[13] A detailed definition of AUGs is given in Section 3.2.2

the ground truth AUG did not harm the validity of the patch. Note that by the manual assessment, we also ensured that the graph comparison done by the heuristic Kuhn-Munkres algorithm [Mun57] was correct, namely, we found no match, which was falsely denoted as equal if it was not. In case ASAP-Repair did not produce a patch, we further examined and documented the reason for non-production.

AndroidCompass+ In the second experiment, we used the AndroidCompass+ dataset. First, we inferred patterns and change rules for each cluster of the protected method calls.

We applied the pattern mining from the MUDetect [ANN+19b] with minimal relative support $min_{support} \in \{0.1, 0.2\}$ and ensured the minimal absolute support was larger than one (i.e., $min_{support} > 1$) to avoid the bias. Recall that this is similar to the experimental setting discussed in Section 6.5.5.

We inferred the change rule *with context* by applying ChaRLI (cf. Section 6.3.2). For the subsequent analysis, we discarded those results from ASAP-Repair with change rule-based repair, which generated a patch by applying a change rule on a misuse present in the method declaration of the same file. This way, we mitigated the influence of the bias of the ground truth fix. Note that results using change rules from other method declaration of the same file and even commit can still be present in the final set.

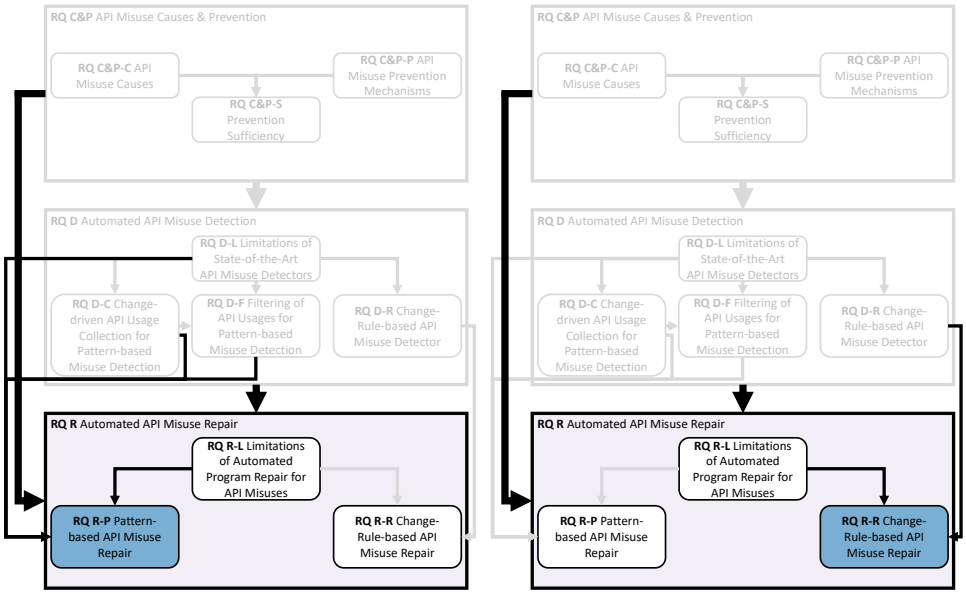
Conceptual Comparison Finally, since ASAP-Repair represents no full-fledged APR technique at the moment, we provide a conceptual comparison to state-of-the-art APR techniques. In detail, we focus on the APR technique applied to API misuses by [KMSH21].

7.5. Validation of ASAP-Repair

In this section, we present the experimental and conceptual validation of ASAP-Repair.

7.5.1. Comparison of Pattern-Based and Change Rule-Based ASAP-Repair

In this section, we provide an experimental comparison of pattern-based and change rule-based repair using ASAP-Repair. Thus, we analyze **RQ R-P** and **RQ R-R**.



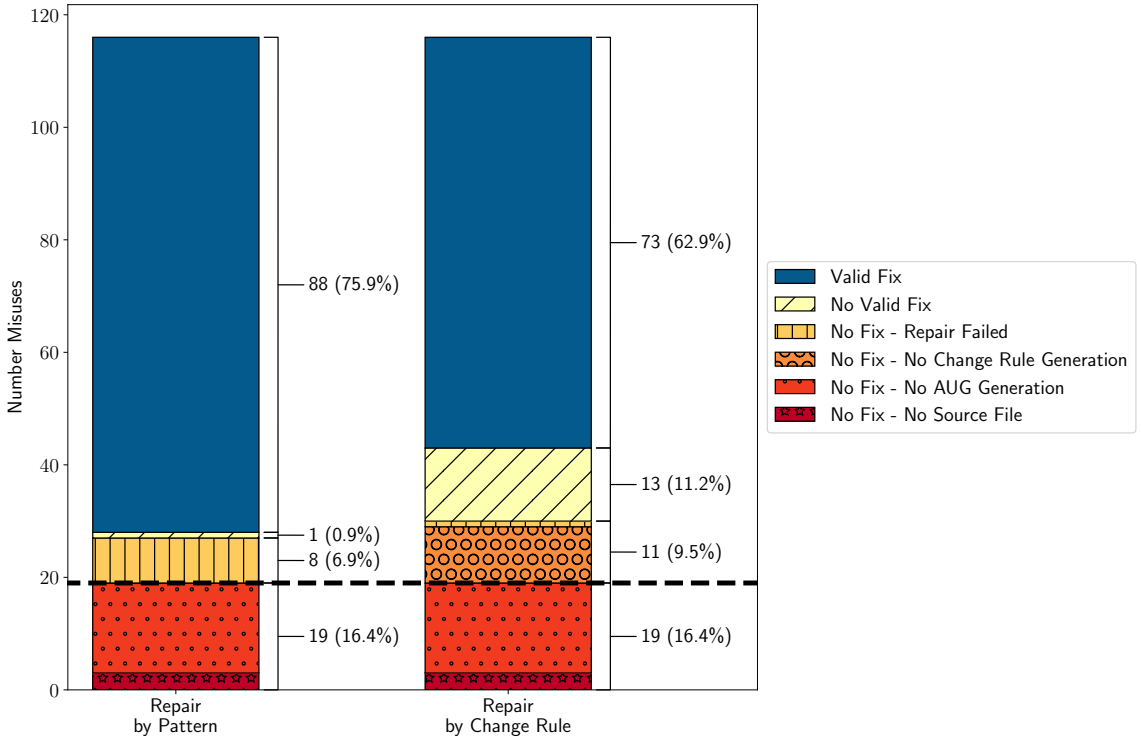


Figure 7.3.: Results of ASAP-Repair on the idealized check using MUBench

Methodology For both experiments discussed in Section 7.4.3, namely, the *idealized check* using MUBench [ANN+16] and the experimental comparison of the pattern- and rule-based variant of ASAP-Repair using the AndroidCompass+ dataset, we prepared scripts to run ASAP-Repair and to analyze its results. In detail, we downloaded all source files of the two datasets, conducted the pattern mining and change rule inference, configured ASAP-Repair to run the pattern- and change rule-based repair, applied the AUG comparison to compute sim_{node} and sim_{edge}, and stored and analyzed the results. For all steps, we provide a replication package[14].

Note that we ran the idealized check on a laptop with the operating system Windows 11 Pro with 64GB RAM and an Intel® Core^{TM} i7-9750H CPU with 6×2.6Ghz. The experiments on AndroidCompass+ were executed on an Ubuntu 20.04.6 LTS system with an Intel® Core^{TM} i7-3930K with 6×3.20GHz. Both settings applied ASAP-Repair with a timeout of *two minutes* for each single repair run.

Results ***Idealized Check MUBench:*** Figure 7.3 depicts the absolute number of obtained patches within the *idealized check on the 116 misuses from MUBench*. We observed that ASAP-Repair with pattern- and change rule-based repair produced a majority of valid fixes while using patterns obtained more patches (i.e., 75.9% of all entries) than change rules (i.e., 62.9% of all entries).

ASAP-Repair failed for 19 entries (i.e., 16.4% of all entries, below the dashed line in Figure 7.3), three of them due to non-downloadable source files and 16 due to non-produced AUGs (i.e., using the implementation from MUDetect [ANN+19b]). For those entries,

[14] http://doi.org/10.5281/zenodo.15594600

ASAP-Repair was not directly responsible for not producing a patch. However, since we could not validate whether a fix would be produced if the AUG was generated, we denoted these cases as *no fix*.

Pattern-based repair failed in eight cases, all of them due to timeouts of ASAP-Repair. For change rules, we found eleven failed cases, ten of which were due to non-produced change rules and one due to a timeout during patch generation. For pattern-based repair, only a single produced patch (i.e., 0.9%) was not valid, while we found that change rule-based repair had 13 (i.e., 11.2% of all misuses) non-valid patches.

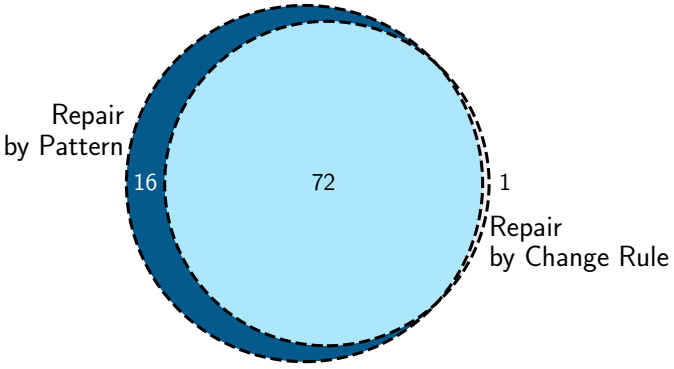


Figure 7.4.: Venn diagram of valid fixes of ASAP-Repair with patterns (left) and with change rules (right) on the idealized check using MUBench

Additionally, we also analyzed whether the pattern- and change rule-based variant of ASAP-Repair found different sets of misuses. Therefore, we depict the number of found patches for each variant in a Venn diagram in Figure 7.4 . We observed that pattern-based repair produced patches for all but one misuse entry compared to the change rule variant (i.e., the only non-valid patch). Thus, we concluded that both techniques were applicable for real-world API misuse repair[15].

Insight R-7 (RQ R-P): Pattern-based ASAP-Repair is applicable in an idealized scenario

We observed that in an idealized situation, as demonstrated in the MUBench idealized check using the fixed version as a pattern, pattern-based ASAP-Repair worked as intended, producing 75.9% *of valid patches.*

Insight R-8 (RQ R-R): Change rule-based ASAP-Repair is applicable in an idealized scenario

We observed that in an idealized situation, as demonstrated in the MUBench idealized check using the misuse and fixed version as change rule inference source, change rule-based ASAP-Repair worked as intended, producing 62.9% *of valid patches.*

Experiment on AndroidCompass+: In comparison to the idealized check, the results on the AndroidCompass+ dataset did not use the ground truth fix but a realistic scenario of previously inferring API usage patterns and change rules.

First, we compared the distribution of the node and edge similarity for all patches obtaining $sim_{node} > 0.9 \wedge sim_{edge} > 0.9$ presented in Figure 7.5 and Figure 7.6. We observed that the mean value among these distributions was larger for the change rule-based variant of ASAP-Repair (i.e., $sim_{node} \approx 96.1\%$ and $sim_{edge} \approx 98.9\%$) than for the pattern-based

[15]Note that these results differ from our previously reported ones [NBKO24], which were obtained by handling depending nodes in the patched AUG

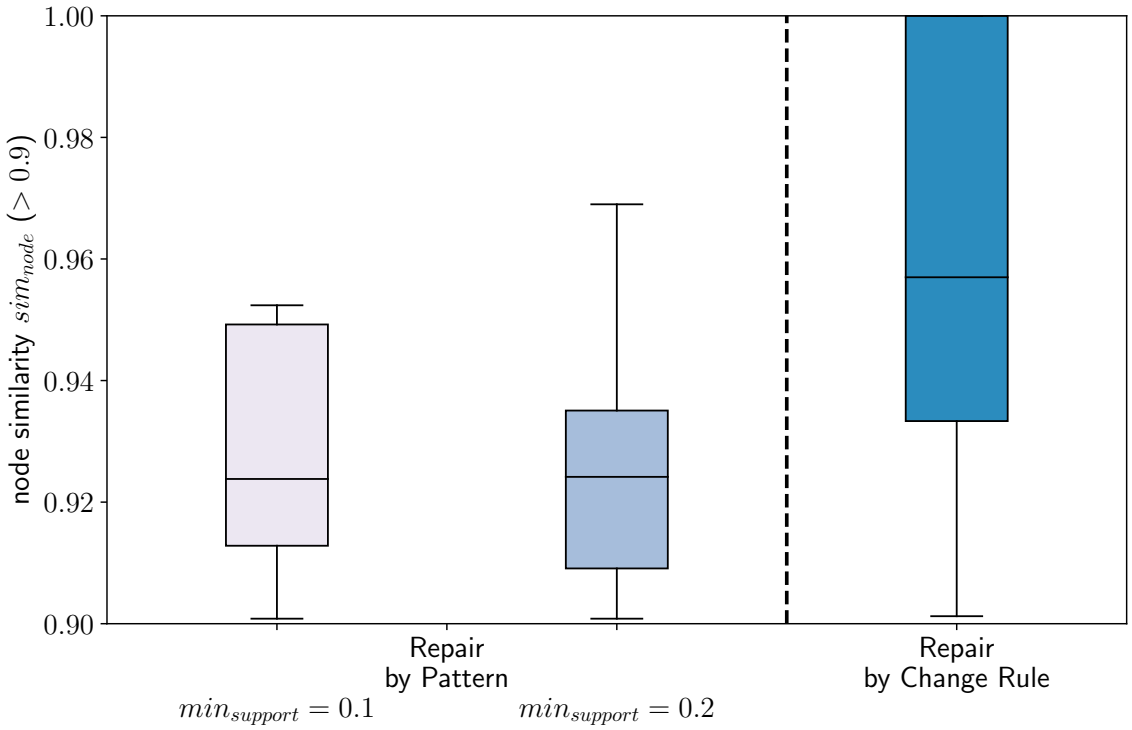


Figure 7.5.: Distribution of node similarity of patches from ASAP-Repair with patterns (left) and change rules (right) using AndroidCompass+ with at least node and edge similarity > 0.9

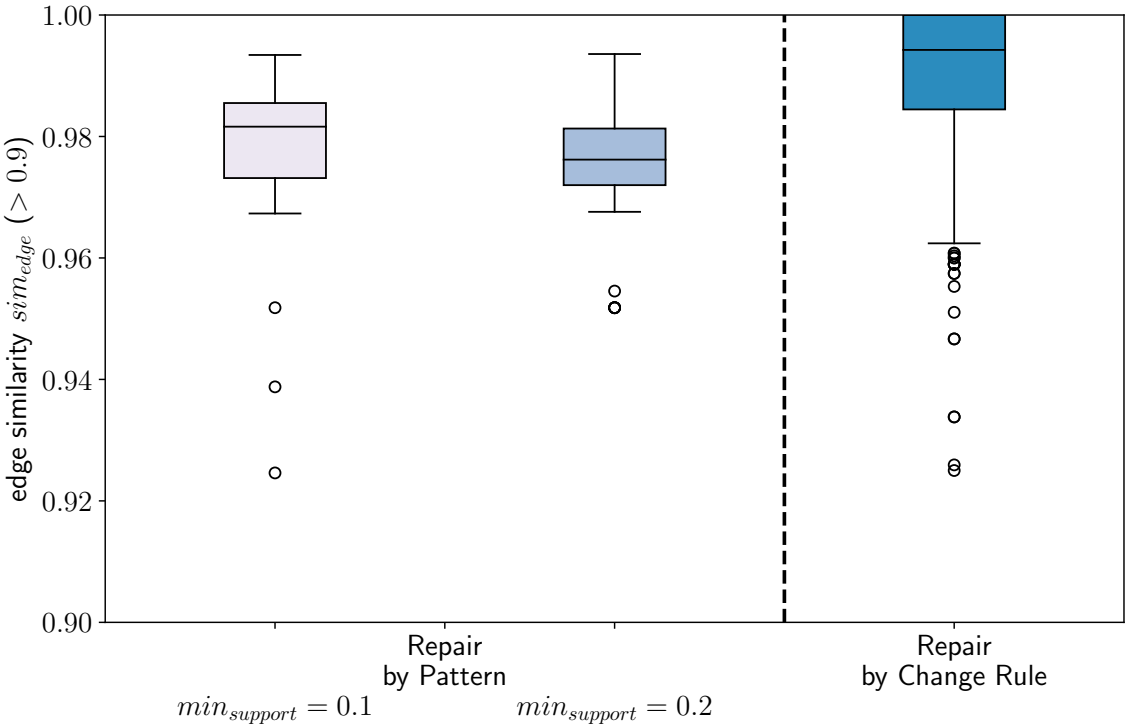


Figure 7.6.: Distribution of edge similarity of patches from ASAP-Repair with patterns (left) and change rules (right) using AndroidCompass+ with at least node and edge similarity > 0.9

one (i.e., $sim_{node} \approx 92.5-92.7\%$ and $sim_{edge} \approx 97.4-97.6\%$). Note that there existed only little differences between the two variants of pattern-based repair using different support values (i.e., $min_{support} = 0.1$ and $min_{support} = 0.2$). All distributions did not stem from the same population. In detail, for $sim_{node} > 0.9$, we found only 26 candidates produced by pattern-based repair with $min_{support} = 0.1$ and 34 candidates with $min_{support} = 0.2$, while the change rule-based had 285 candidates.

We further examined the absolute number of produced patches in Figure 7.7. Particularly, we depict the number of patches satisfying both similarities sim_{node} and sim_{edge} constraint (i.e., larger or equal to an increasing threshold) for each repair variant. We concluded that in all considered cases, the change rule-based ASAP-Repair produced more patches with larger similarity than the pattern-based variant.

Having $sim_{node} = sim_{edge} = 1$ denoted a perfect match of the generated patched AUG produced by ASAP-Repair in comparison to the ground truth and, thus, a *valid patch*. Based on Figure 7.7, only change rule-based ASAP-Repair produced valid patches in 102 cases, while both pattern-based variants failed to find any valid patches at all.

For those patches found by the rule-based repair, we also checked whether our misuse detector Change Rule-based API Misuse Detection (RuDetect) is able to *detect* these misuses using the change rule applied for repair. In detail, based on our experiments presented in Section 6.5.5 (cf. Comparison of RuDetect with state-of-the-art), we used the similarities `APIIndicatorExasVector` and `ExasVectorCosine` with most restrictive *threshold* $= 0.7$ for applicability check (cf. Equation 6.1 in Section 6.3.3). For both variants, we found that 91 *out of the* 101 *patched misuses* are correctly detected as API misuse. Thus, in a full-

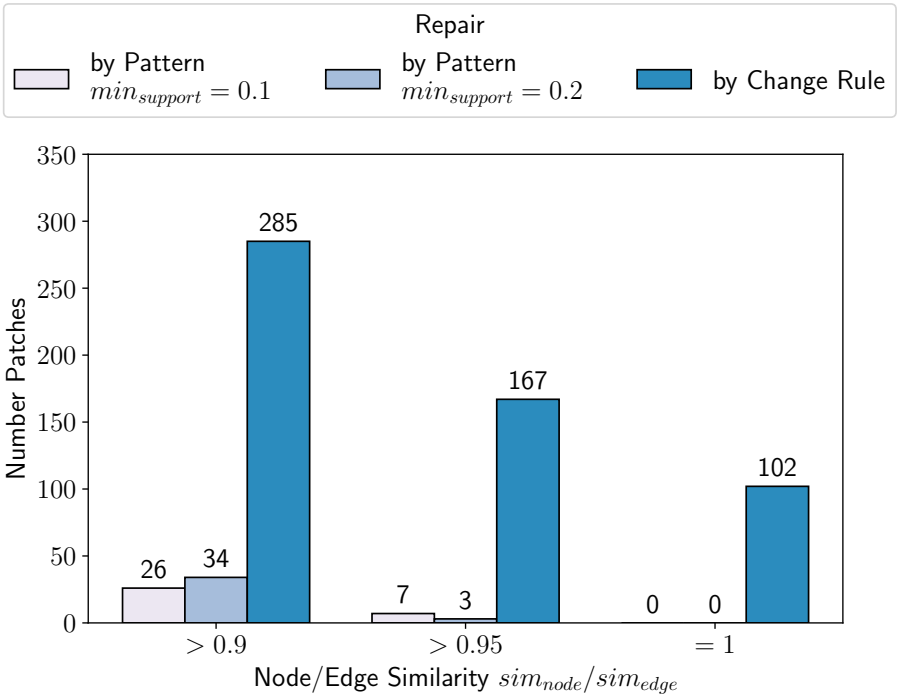


Figure 7.7.: Number of patches from ASAP-Repair with an simultaneously increasing sim_{node} and sim_{edge} threshold to the ground truth fix using AndroidCompass+

stacked implementation of ASAP-Repair with previous misuse detection using RuDetect 91 misuses would have been fixed.

Insight R-9 (RQ R-P): Pattern-based ASAP-Repair does not produce valid patches in a realistic scenario

In a realistic scenario, as demonstrated in the AndroidCompass+ experiment, pattern-based ASAP-Repair (1) did not produce valid patches, and (2) had only a small number of similar patches compared to ground truth fixes.

Insight R-10 (RQ R-R): Change rule-based ASAP-Repair produces valid patches in a realistic scenario

In a realistic scenario, as demonstrated in the AndroidCompass+ experiment, change rule-based ASAP-Repair (1) produced 102 valid patches, 91 of them also found by our misuse detector RuDetect, and (2) had a larger number of similar patches compared to ground truth fixes and also a larger number of patches than the pattern-based variant of ASAP-Repair.

No Fixes for pattern-based ASAP-Repair: These results partially contradicted the observation from the idealized check, in which pattern-based ASAP-Repair performs better than the change rule-based variant. The reason was that the patterns used in the idealized check are AUGs representing full method declarations almost perfectly matching

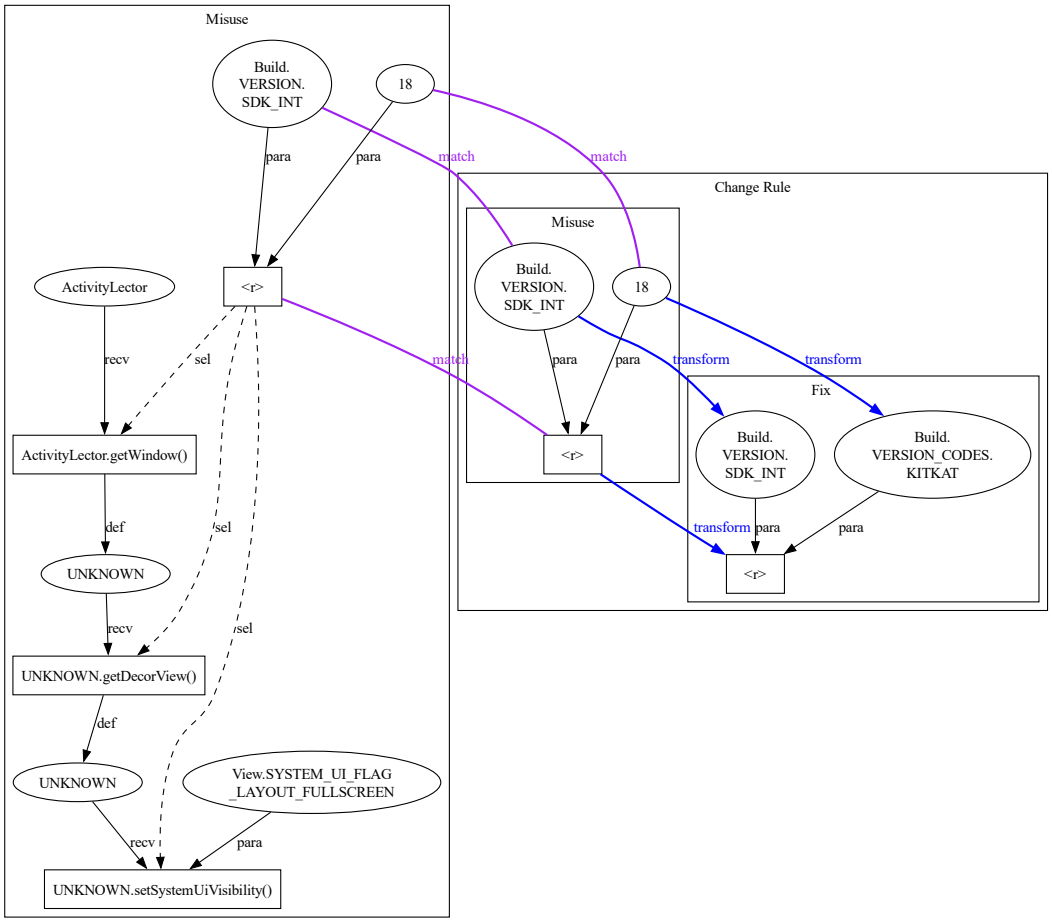


Figure 7.8.: Sample of a matching between misuse (left) and change rule (right) from a successful patch

the misuse present. When mining patterns, we used the same set of patterns as found in the AndroidCompass+ experiment discussed for API misuse detection (cf. Section 6.5.5). From this experiment, we already knew that using RuDetect performd better than pattern-based misuse detection using MUDetect (cf. **Insight D-20**). Recall that this was caused since the patterns were less related to the checked misuses (i.e., Android compatibility checks). Thus, it was not surprising that pattern-based repair could not produce any valid patch.

Qualitative View on Change Rule-Based ASAP-Repair: We further investigated the good performance of change rule-based ASAP-Repair by qualitatively analyzing the patches together with their matching between the misuse AUG and the change rule. For that purpose, analyzed the matching of a valid patch represented in Figure 7.8. We selected this example since we found it representative of a set of found patches as well as sufficiently succinct to be depicted in this thesis[16]. Note that the `<r>` node denotes an action node representing an `if` condition, which protects the action nodes connected via the dashed

[16]Other examples can be found in our replication package

control flow edges labeled with `sel`.

We tracked the origin of this misuse as well as the change rule depicted in this example. In this case, misuse and rule shared the same commit but originated from two different method declarations[17]. In detail, this commit updated the compatibility check from SDK level 18 to 19 (i.e., version *KitKat*), which enabled a new immersive mode for Android apps, which could hide the navigation bar of the Android operating system (cf. Android release note[18]). Previously to *KitKat*, this behavior was obtained by hiding and showing the navigation bar using the `setSystemUIVisibility` method. This behavior, however, was still supported in KitKat using the new immersive view.

Apart from the discussion of whether the sample in Figure 7.8 essentially represents an API misuse and whether it is valid to apply change rules from the same commit, we focused on a different aspect. When considering the change rule, it was too succinct since it only represented the `if`-condition without its protected methods. This way, the matching, while being correct, only worked by chance, namely, updating the object node representing the SDK level (i.e., `18`) to the correct one.

We observed this situation in other patches as well. This result implicated that while change rule-based ASAP-Repair works correctly and as intended in this situation, it might fail in other situations. For instance, assume a similar change rule as in Figure 7.8, which updates the SDK level to 30, a level at which the method `setSystemUIVisibility` was deprecated[19]). Thus, at the current state, we assess change rule-based ASAP-Repair due to such issues as a partially applicable and reliable technique in practice.

Insight R-11 (RQ R-R): Change rule-based ASAP-Repair is only a preliminary technique for API-specific automated program repair

While producing valid patches, we observed in a qualitative review that change rules could lack sufficient context. Thus, further research should strive to increase the resilience of change rules to foster the quality of generated patches.

Implications We found that, in principle, both variants of ASAP-Repair, namely *pattern-* and *change rule-based*, were applicable. However, pattern-based ASAP-Repair failed more often since it could not extract useful patterns compared to the inference of change rules. We hypothesize that the *correct selection of fixing commits* for change rule inference is the 'main ingredient' to the success, as seen with RuDetect for API misuse detection as well as in change rule-based ASAP-Repair. While currently, the selection of commits is meant to be manual (cf. Section 6.3.1), further research may identify fixing commits automatically. This mechanism may go beyond keyword-based techniques on commit messages for finding fixing commits as part of the well-known *SZZ algorithm* by [SZZ05] to identify error-introducing commits. Qualitatively, we determined necessary improvements of change rules. Currently, too small change rules may cause arbitrary correct results. Further research has to strive for robust change rules and subsequent patches, which would represent a strong APR technique for practical usage.

[17] cf. the related GitHub project https://github.com/raulhaag/MiMangaNu/commit/86c09774914d7242a04427f1f95acff7a2a2e726 last accessed: 2025/02/24

[18] https://developer.android.com/about/versions/kitkat#44-beautiful-apps last accessed: 2025/02/24

[19] cf. Android documentation at https://developer.android.com/reference/android/view/View#setSystemUiVisibility(int) last accessed: 2024/02/24

Table 7.1.: Conceptual Comparison of ASAP-Repair to API-specific, behavioral-based APR techniques.

	ASAP-Repair (Ours)	APIFix [GRS+21]	BugMem [KPW06]	CDRep [MLLD16]	GenPatch [Wei06]	Pachika [DZM09]	QACrashFix [GZW+15]	TADEF [BOST22]
Available Software Artifact	●	●	○	○	○	●	◐	○
Test-free Fault Localization	●	●	●	●	●	○	◐	●
Test-free Patch Validation	◐	◐	◐	◐	◐	○	◐	○
Generic Repair Patterns	●	●	●	○	●	◐	●	○
Generic API Misuses Types	●	○	○	○	●	●	●	○
Independence of External Sources for Patch Generation	○	○	●	●	○	●	○	●

7.5.2. Conceptual Comparison of ASAP-Repair to State-of-the-Art APR

Classification of ASAP-Repair As previously shown in Figure 7.1 (cf. Section 7.2.2), we denote it as an *API-specific, behavioral-based APR technique*. In detail, it is *behavioral-based* since it aims to change the source code and, thus, the behavior of the code. Moreover, it is *API-specific* since it specifically targets API misuse by applying API-specific patterns representing frequent and likely correct API. Moreover, it can be considered as a *learning- or pattern-based APR* since, for pattern-based repair, correct API usage is learned in the form of frequent patterns.

Comparison to Other API-Specific, Behavioral-Based APR Due to its classification first, we compared ASAP-Repair to all API-specific, behavioral-based techniques, namely those shown in the upper right quadrant of Figure 7.1 in Section 7.2.2. We depict an overview of this comparison in Table 7.1.

Available Software Artifact: We provide ASAP-Repair as an available artifact[14]. We found an available, linked artifact also for APIFix [GRS+21] and Pachika [DZM09]. For QACrashFix [GZW+15], the authors provided a URL, but the link did not work anymore. TADEF [BOST22] linked their experimental data, however, not the repair technique. For BugMem [KPW06] and CDRep [MLLD16], no artifact was linked, even though experiments were conducted. GenPatch [Wei06] only states the general notion of a repair technique without providing a direct evaluation or a software artifact.

Test-Free Fault localization: We analyzed whether those APR techniques required tests for fault localization (i.e., first row in Table 7.1) or patch validation (i.e., second row in Table 7.1). We observed that ASAP-Repair aligns with most techniques that did not require tests for fault localization. Particularly, APIFix [GRS+21] applied transformation

rules from breaking changes, BugMem [KPW06] leveraged previous fixes from the project's VCS, CDRep [MLLD16] and TADEF [BOST22] used manually crafted repair templates, and GenPatch [Wei06] similar to ASAP-Repair applied specifications mined from source but in opposite to us they inferred state machines. While QACrashFix [GZW+15] did not directly define a dependency on tests, it was based on crash reports, which required the execution of the program to determine the fault. Pachika [DZM09] relied on test cases for both fault localization and patch validation. Particularly, it inferred object behavior differences from execution traces of passing and failing tests and checked the patch with the same test suite.

Test-Free Patch Validation: Regarding the patch validation next to Pachika, TADEF also denoted a test-based patch validation. However, since they targeted API misuses of the DL framework TensorFlow, they also measured other factors, such as the training performance of fixed models. For all other techniques, including ours no tests were used for the evaluation. Nevertheless, we assumed that test-based validation was reasonable for them and us in practice. For instance, CDRep, GenPatch, and QACrashFix assumed that experts or developers validated the patches (e.g., in a code review-like process). APIFix, BugMem, and our ASAP-Repair assessed the patch quality by measuring similarity to the ground truth fix, which could not available in practice. Thus, both variants were either too cumbersome or not possible at all. Regarding the industrial application of APR (cf. [NHL18, BSPC19, MBC+19, KWM+21]), we suggest test-based validation with a subsequent manual validation.

Generic Repair Patterns: ASAP-Repair has no restriction to repair patterns similar to APIFix, BugMem, GenPatch, and QACrashFix. Instead, CDRep and TADEF, as stated before, relied on manually crafted repair templates and thus did not generalize. Pachika added either preconditions to methods or deleted methods at all, which hardened update operations.

Generic API Misuse Types: Moreover, ASAP-Repair does not have any restrictions regarding API misuse types or domains, similar to GenPatch, Pachika, and QACrashFix. In contrast, CDRep (i.e., targeting cryptographic APIs) and TADEF (i.e., targeting TensorFlow) focused on specific misuse patterns. While not limiting the domain, APIFix (i.e., focuses on breaking changes) and BugMem (i.e., only targets project-specific bugs) had restrictions regarding the API misuse scope.

Independence of External Sources for Patch Generation: ASAP-Repair requires external data, namely, other API usages or fixes to infer patterns or change rules. Similarly, GenPatch required other API usages to infer state machines. APIFix relied on the source code changes in a library, while QACrashFix used Q&A posts from StackOverflow. In contrast, BugMem and Pachika relied on code features from the same project. CDRep and TADEF had static patch templates and did not need any additional information.

Comparison to Other Generic, Behavioral-Based APR Based on the experiments by Kechagia et al. [KMSH21], we know that, while limited, generic, behavioral APR is applicable for API misuses. However, all techniques executed by the APIARTy framework [KMSH21] require a test-based fault localization, which has been found to be a major issue for not producing patches. Note that Avatar [LKKB19b], TBar [LKKB19a], and SimFix [JXZ+18] were not directly executed in their experiment. However, their fault localization also requires tests. Even though not included in the experiment by Kechagia et al. [KMSH21], the constraint-based patch generation introduced by Angelix [MYR16] also

requires tests for fault localization.

Thus, next to its API specficity, ASAP-Repair requires no tests for fault localization compared to these techniques.

Comparison to Other Learning-Based APR We, particularly, compare to learning-based APR since they conduct a similar procedure to ASAP-Repair by using knowledge from previous fixes and correct usages. However, learning-based techniques, particularly DL ones, require large datasets as well as huge computational effort. Even if pre-trained models are used, experimental evaluation becomes more complicated due to the data leakage issue, particularly if training data is unknown and the trained model is used as a black box.

In contrast, ASAP-Repair, especially its rule-based repair, has a far simpler inference technique. Even though it can suffer from data leakage (cf. Section 7.5.4) as well, the training data is transparent, allowing previous filtering of ground truth fixes. Moreover, as discussed by Zhang et al. [ZFM+23], many learning-based APR techniques also require tests for fault localization.

Comparison to Other State-Based APR Compared to state-based APR techniques, our technique ASAP-Repair does not target the liveness criteria. Thus, state-based APR is meant for software operation, particularly for critical software components. Instead, ASAP-Repair aims to support software developers to permanently fix API misuses.

Insight R-12 (RQ R-P): Conceptual benefits of pattern-based ASAP-Repair for API misuses to state-of-the-art APR

We found that pattern-based ASAP-Repair had conceptual benefits compared to state-of-the-art APR for API-specific repair due to (1) specifically targeting API misuses, (2) no dependency on tests for fault localization, and (3) producing a permanent solution.

Insight R-13 (RQ R-R): Additional conceptual benefits of change rule-based ASAP-Repair for API misuses comparison to state-of-the-art APR

In addition to the benefits of pattern-based ASAP-Repair, rule-based ASAP-Repair had less computational effort compared to state-of-the-art APR, particularly learning-based techniques using DL.

7.5.3. Towards Code Patches from ASAP-Repair

Currently, we applied ASAP-Repair solely on AUGs to assess its repair ability by avoiding parsing and compiling issues. As seen in our results, the patched AUGs generated by ASAP-Repair could create many irrelevant patches, namely, in case of AndroidCompass+ a set of compatibility checks that are not linked to the protected method. Thus, a full-fledged APR technique, creating patched source code, was not meaningful at the moment.

Nevertheless, in case its repair abilities can be improved, it requires a full-fledged technique to produce patched source code. Therefore, we briefly discuss two possible variants to derive source code from patched AUGs together with their technical hurdles to overcome:

1. Inference of modifications of Abstract Syntax Trees (ASTs) based on AUG transformations obtained from ASAP-Repair;
2. Direct transformation of the patched AUG into source code.

Inference of AST Modifications The notion of the first variant is to extract AST modification operations based on the obtained AUG transformation. In detail, this encompasses steps to *determine single AST modifications* (e.g., adding an `if` AST-node), to align them in an *order to resolve dependencies* among them (e.g., a new parameter of a method call requires a previous initialization), as well as to *include necessary modifications*, for instance, when moving a method call into a `then` block necessary data dependencies have to be resolved. The benefit of this procedure would be that one can directly modify the AST of the buggy code, which has been obtained when generating the AUG. This modified AST can be directly transformed into patched source code.

We tested this variant and found that for realistic code patches, we required a variety of single AST modifications to create valid patches. For instance, assume that we use a return value of a method call, which should be protected by an `if`-condition. In case we add this condition, we also have to protect the subsequent usage of the return value or add a default initialization of this value. In general, single code modifications can trigger many subsequent data and control flow-related implications, which increases the number and variance of required AST modifications.

Transformation from AUG to Source Code The second variant aims to use the patched AUG and transform it directly into source code. A naïve solution for this variant is a topological sorting of AUG nodes, obtaining possibly multiple sequences of code statements. This sorting is always possible since AUGs are directed acyclic graphs (cf. Szpilrajn [Szp30] as cited by Knuth [Knu97, p. 268]). Based on the sequences, one can reconstruct the source code elements, such as the handling of different cases (e.g., `if`-statements) or errors (e.g., `try-catch-finally`-blocks), as well as special code blocks (e.g., `synchronized`). In the final step, undefined variables have to be added either as method parameters in the respective declaration or as global variables, as well as missing dependencies (i.e., `import`s).

Currently, this approach is not efficient since there can exist many possible code variants due to the abstraction in the AUG. This abstraction causes unknown parameter order for method calls (i.e., due to unsorted `para` edges) or unclear branching in `if`-statements (e.g., whether a method call is part of the `then`- or `else`-branch). Moreover, AUG generation can lack type information, thus generating `UNKNOWN` nodes in an AUG. Then, we cannot generate syntactically correct source code. Note that AUGs relate to the method declaration scope, and thus, the changes may interfere with code components beyond this scope.

Therefore, we suggest a formalized transformation from ASTs to AUGs and vice versa as a bijective function. However, currently, there exists only a single directed transformation from ASTs to AUGs, which is only implicitly determined in the MUDetect artifact [ANN+19b][20]. Thus, further research has to formalize these steps and derive an efficient transformation.

7.5.4. Threats to Validity

We finally consider the threats to validity of our experiments, which may limit the significance of our results. Again, we consider the *internal* and *external validity* according to Siegmund et al. [SSA15].

[20] cf. implementation at https://github.com/stg-tud/MUDetect last accessed: 2025/03/14

Internal Validity We implemented our concept of ASAP-Repair as a prototypical implementation, which might contain errors. This issue also concerned related tools and libraries applied and extended for evaluation, for instance, MUDetect [ANN+19b] used for AUG generation and API usage pattern mining. This issue might negatively affect results, and thus, for replicability reasons, we published the source code of ASAP-Repair together with related analysis scripts and datasets in our replication package[14].

Currently, ASAP-Repair does not create source code patches. Thus, we did not investigate whether generated patches compiled or not. Therefore, the number of valid patches might decrease.

Another issue was related to the dataset. For instance, the ground truth fixes in MUBench and AndroidCompass+ to which we compared our generated patches could be wrong or represented non-sufficient solutions. Thus, the reported number of valid fixes might decrease as well.

We avoided in the experiment with AndroidCompass+ data leakage by using either larger support for pattern-based repair or filtering out the ground truth change rules for rule-based repair. However, due to forked projects within AndroidCompass+, still identical fixes might exist. Moreover, our evaluation might suffer from temporal bias, namely, change rules and patterns might not be available at the time of the misuse. This bias can cause some patches to only be generated later or not at all in practical scenarios. Since we determined rule-based repair of ASAP-Repair as a functional but still preliminary technique (cf. **Insight R-11**), we abandoned techniques to cope with temporal bias for future analysis.

Similar to the issues described in Section 6.5.6, we applied FPM to obtain API usage patterns using the complete source files with all method declarations. Since FPM required a sufficiently large $min_{support}$ (cf. Section 3.5) to obtain patterns efficiently (i.e., due to memory- and time restrictions), certain patterns that were able to fix the misuse would not have been detected. In contrast, change rules were inferred from single method declarations and thus were more likely to contain a matching fix if it was present in our dataset. Thus, rule-based repair might be positively biased towards better matching templates for repair.

External Validity Our datasets represent one threat to external validity. In detail, AndroidCompass+ only contained `if` conditions of Android compatibility checks. For validation of whether ASAP-Repair is applicable to different kinds of misuses (e.g., false order or redundant calls of API methods) and domains (e.g., cryptography, graphical user interface), we require further and large datasets.

Moreover, we did not experimentally compare to state-of-the-art techniques due to the lack of source code patches by ASAP-Repair. Thus, currently, we cannot determine the beneficial effect in terms of real-world bug fixes compared to state-of-the-art APR. We discussed in Section 7.5.3 two variants to obtain source code patches in case the fix quality on AUGs is further improved.

Currently, AUGs are restricted to the Java language with object-oriented features and intra-procedural API misuses (i.e., within a single method declaration). This way, changes to the data structure of AUGs are necessary to cope with other programming languages and paradigms as well as further techniques to cope with inter-procedural patches.

As seen in practical applications of APR, it requires further steps for quality assurance, such as independent tests or code reviews [BSPC19, MBC+19], most likely since automatically generated patches are not trusted by the developers. Thus, we suggest that the practical application of ASAP-Repair requires further quality assurance steps as well.

7.6. Summary API Misuse Repair

Summary In this chapter, we targeted **RQ R** on how API-specific information could support its Automated Program Repair. For this purpose, we analyzed the limitations of current APR techniques in **RQ R-L**. In detail, we reviewed the results of previous APR survey articles and studies [Mon18a, Mon18b, KMSH21, ZFM+23] as well as a research community webpage[21] and summarized the limitations of APR techniques when repairing API misuses.

Then, we introduced the concept of an API-specific APR technique named *API-Specific Automated Program Repair (ASAP-Repair)*. It operates in two variants, namely, *pattern-based* (cf. **RQ R-P**) and *rule-based repair* (cf. **RQ R-R**).

Pattern-based repair leverages *API usage patterns* used by pattern-based misuse detectors such as MUDetect [ANN+19b]. These patterns represent frequent sub-graphs of AUGs (e.g., aug_p). The pattern-based repair assumes that the pattern aug_p detected a misuse in aug_u (i.e., representing another API usage) as a violation of this pattern (e.g., using MUDetect [ANN+19b]). Then, the pattern-based repair *matches the nodes of the misuse AUG* aug_u *and the pattern* aug_p. Based on this match, ASAP-Repair *extracts transformation operations*, which denote how to change aug_u to *construct a patched AUG* $aug_{u'}$.

For *rule-based repair*, we applied *change rules* already introduced in Chapter 6, representing the transformation of a previous API misuse fix as AUGs in the form of $aug_{m'} \rightarrow aug_{f'}$. The rule-based repair produces a match between the nodes of the misuse AUG (i.e., aug_u) and those of the misuse part of the rule (i.e., $aug_{m'}$). Based on this match, aug_u is edited according to the transformation described by the change rule (i.e., transforming nodes and edges from $aug_{m'}$ to $aug_{f'}$). By applying these transformations on aug_u, the rule-based repair produces a patched AUG $aug_{u'}$.

We implemented *a prototype of ASAP-Repair*[14] and evaluated its functionality by applying an idealized check using the *MUBench* dataset [ANN+16] and a more realistic scenario with our own dataset *AndroidCompass+*, an extension of our previous dataset AndroidCompass [NBKO21b]. In detail, we assessed patches as *valid* if they match the ground truth AUG provided by both datasets. Since ASAP-Repair currently works on an AUG level, we provide *a conceptual comparison of ASAP-Repair to state-of-the-art APR* techniques.

Contribution RQ R-L We found that many state-of-the-art APR techniques relied on test suites for fault localization, a mandatory step before patching a bug (cf. **Insight R-1** 'Dependency of APR on test suites' on page 208). Moreover, research on APR relied on too few and too small benchmarks of API misuses (cf. **Insight R-2** 'Few and too small APR benchmarks for API misuses' on page 209). Thus, we reused our AndroidCompass+ dataset already introduced in Chapter 6 (cf. Section 6.4.1) to target this limitation. Based on the classification of APR, *state-based APR* (e.g., changing the memory state to overcome bugs at runtime) only provides temporal solutions (cf. **Insight R-3** 'Limitations of state-based APR techniques as a temporal solution with possible side-effects for API misuses' on page 211) for bugs and API misuses in particular. Instead, *behavioral APR* changes the buggy behavior permanently, typically by changing the source code. However, most generic techniques in this class (i.e., those without particular focus on APIs) fail to generate patches since their generic patch mechanism does not match the specific nature of API misuses or they fail to localize the API misuse, which is typically done via tests (cf. **Insight R-4**

[21] https://program-repair.org last accessed: 2025/03/17

'Non-sufficient solution for API-specific APR by traditional, generic, behavioral-based APR techniques' on page 212). While machine learning techniques were successfully applied for APR tasks, we found that these required large datasets, encompassed a large effort for training, or previous pre-trained could suffer from data leakage, namely, containing the ground truth fixes of known APR benchmarks (cf. **Insight R-5** 'Promising but still limited applicability of learning-based, generic, behavioral-based APR techniques for API misuses' on page 213). Finally, we reviewed API-specific, behavioral APR and found that these are limited regarding their application domain (e.g., cryptography or machine learning), require high-quality input, and suffer from computational expensive fault localization (cf. **Insight R-6** 'Too narrow applicability of API-specific, behavioral APR for API misuses' on page 214).

Thus, we answer **RQ R-L** as follows:

RQ R-L What are the limitations of state-of-the-art APR techniques to repair API misuses?

We found that state-of-the-art APR techniques did not sufficiently target API misuses due to (1) non-sufficiently large benchmarks of API misuses, (2) their dependency on tests for fault localization, (3) their non-sufficient patch mechanism for API misuses, and (4) their limitation to too specific APIs and their misuses.

Contribution RQ R-P Our technique ASAP-Repair with its *pattern-based repair* mechanism repaired – in principle – API misuses, namely, finding 75.9% patches in the idealized check (cf. **Insight R-7** 'Pattern-based ASAP-Repair is applicable in an idealized scenario' on page 224). However, we observed that in a more realistic scenario, no valid patches were produced (cf. **Insight R-9** 'Pattern-based ASAP-Repair does not produce valid patches in a realistic scenario' on page 227). We found that this was mainly due to the inability of the FPM to infer patterns able to repair a certain misuse. However, pattern-based repair is a promising approach, particularly since it overcomes the limitation of the dependency on test suites for fault localization (i.e., **Insight R-1**). Assuming further improvements in pattern inference (e.g., in pattern-based API misuse detection), we consider it a strong alternative to state-of-the-art API APR (cf. **Insight R-12** 'Conceptual benefits of pattern-based ASAP-Repair for API misuses to state-of-the-art APR' on page 232). In comparison to the rule-based variant (cf. presented in the next paragraph), it does not require a prior fix of a misuse. Moreover, we provided further steps to obtain source code patches from the patched AUGs, making it an alternative for state-of-the-art APR in Section 7.5.3.

Thus, we answer **RQ R-P** as follows:

RQ R-P Do API usage patterns provide benefits for API-specific APR?

We determined that API usage patterns are valuable information to improve API-specific misuse repair since it is conceptually superior to traditional APR as it does not require test-based fault localization and it specifically targets APIs without domain and bug type restrictions. While we demonstrated that our pattern-based ASAP-Repair was able to patch API misuses, its current bottleneck is the lack of high-quality pattern inference. Thus, currently, pattern-based repair is limited in its practical application.

Contribution RQ R-R In its second variant, ASAP-Repair applied *rule-based repair*, leveraging change rules, which could be used by our misuse detector RuDetect for detection (cf. Chapter 6). We demonstrated its applicability in the idealized check with 62.9% of valid patches (cf. **Insight R-8** 'Change rule-based ASAP-Repair is applicable in an idealized scenario' on page 224) as well as in the practical scenario with 102 valid patches, from which 91 were detectable by RuDetect (cf. **Insight R-10** 'Change rule-based ASAP-Repair produces valid patches in a realistic scenario' on page 227). This way, it performed better than its *pattern-based* counterpart. However, in a qualitative review of the patches of the realistic scenario, we found that some change rules lack sufficient context and had the potential to produce invalid fixes (cf. **Insight R-11** 'Change rule-based ASAP-Repair is only a preliminary technique for API-specific automated program repair' on page 229). Compared to state-of-the-art APR rule-based ASAP-Repair specifically targets API misuses, requires no tests for fault localization, produces a permanent solution, and due to its automated change rule inference (cf. ChaRLI in Section 6.3.2) has less computational effort than, for instance, other promising learning-based APR techniques using machine learning (cf. **Insight R-13** 'Additional conceptual benefits of change rule-based ASAP-Repair for API misuses comparison to state-of-the-art APR' on page 232).

Thus, we answer **RQ R-R** as follows:

RQ R-R Do change rules of previous API misuse fixes provide benefits for API-specific APR?

We found change rules to be a fruitful source to target API-specific APR, particularly by conceptualizing and implementing rule-based repair within ASAP-Repair. This way, it is not only conceptually superior to state-of-the-art APR (i.e., API specificity, no test-based fault localization, less computational effort) but also outperforms its pattern-based variant. While still limited in its practical applicability, we consider it a strong competitive for API-specific program repair.

8 Conclusion

In this chapter, we summarize the main chapters of this thesis, namely, chapters 4, 5, 6, and 7 (cf. Section 8.1) and answer the main research questions by reviewing the main results and contributions (cf. Section 8.2). Finally, we explore potential extensions and future research directions based on this thesis in Section 8.3.

8.1. Summary of the Thesis

In this thesis, we targeted the general problem of Application Programming Interface (API) misuses. API misuses denote deviant usages of an API than expected by the API developers (i.e., those who developed a library and the API) by client developers (i.e., those who use the library in their application code) causing a negative software behavior (e.g., crashes or security vulnerabilities). After introducing the topic (cf. Chapter 1), the detailed research questions (cf. Chapter 2), and the necessary fundamentals and background (cf. Chapter 3), we presented our research contributions.

API Misuse Root Causes and Prevention (cf. Chapter 4) Based on this central topic, we considered the root causes and potential prevention mechanisms for these causes.

In detail, we conducted two subsequent Systematic Literature Reviews (SLRs) mapping empirical studies analyzing root causes (i.e., 65 studies) and prevention mechanisms (i.e., 411 studies). By using qualitative research methods, we elicited root causes and mapped them to the applied research methods of the surveyed studies. This way, we assessed the scientific evidence and provided an overview of the research effort of different root causes.

Subsequently, we inferred prevention mechanisms and mapped them to previously extracted root causes together with their level of evaluation. Hence, we determined potential research opportunities regarding studies on specific root causes as well as prevention mechanisms targeting these root causes.

Improving Pattern-Based API Misuse Detection (cf. Chapter 5) Then, we analyzed pattern-based API misuse detection. This misuse detection use inference techniques, such as Frequent Pattern Mining (FPM) on API usages from other client code, to obtain API usage patterns. Then, the detectors assess violations of these patterns as API misuses.

First, we surveyed state-of-the-art API misuse detectors using an SLR and summarized their techniques and results together with their limitations. For further analysis, we selected the misuse detector MUDetect [ANN+19b] to evaluate our improvement techniques. MUDetect represents API usages and its pattern as graphs denoted as API Usage Graphs (AUGs).

Based on the limitations of misuse detectors, we developed a technique leveraging change information from commits editing API usage in client code. In detail, this technique, denoted as *Relevant API Information Extractor (RAIX)*, automatically extracts API-related keywords from such a commit and subsequent search and filter strategies. This way, we found similar API usage examples by code search auxiliaries either in the same project (i.e., *internal*) or in other code (i.e., *external*) and conducted different filter strategies.

We evaluated the effect of applying RAIX and the search and filter strategies with multiple configurations (e.g., search locations and filtering techniques) together with the MUDetect using two different misuse datasets, namely MUBench [ANN+16] and AU500 [KL21].

Change Rule-Based API Misuse Detection (cf. Chapter 6) In Chapter 6, we introduced the concept of *change rules*, describing the essential changes of an API misuse fix. These rules denote two connected AUGs as $aug_{m'} \rightarrow aug_{f'}$ representing the changes conducted in the misuse graph $aug_{m'}$ to obtain the respective fix graph $aug_{f'}$.

We introduced a technique named *Change Rule Inference (ChaRLI)*, which – with minimal manual effort – automatically extracts change rules from a commit. Moreover, we developed a new misuse detector named *Change Rule-based API Misuse Detection (RuDetect)* that integrates ChaRLI. RuDetect applies change rules on potential misuses using an *applicability check*, and if a rule is denoted applicable, we use a *similarity metric*, *sim*, to decide whether an API usage is a misuse. In detail, an aug_u is a misuse according to the change rule $aug_{m'} \rightarrow aug_{f'}$, if aug_u is more similar to the misuse part of a rule $aug_{m'}$ than to the fix part $aug_{f'}$, namely, $sim(aug_{m'}, aug_u) > sim(aug_{f'}, aug_u)$.

We evaluated RuDetect with 16 different graph similarity metrics and two applicability check variants based on three different misuse datasets (i.e., MUBench, AU500, and our own dataset AndroidCompass+) and compared it with the pattern-based misuse detector MUDetect.

Towards API Misuse Repair (cf. Chapter 7) We further analyzed in Chapter 7 how detected API misuses can be repaired automatically. For that purpose, we surveyed *state-of-the-art Automated Program Repair (APR) techniques* and considered their limitations when fixing API misuses.

We targeted these limitations by applying patterns and change rules from the previous API misuse detection as repair templates. In detail, we implemented this idea as an artifact named *API-Specific Automated Program Repair (ASAP-Repair)*, a technique that repairs API misuse as AUGs. In detail, ASAP-Repair operates in two variants: a *pattern-* and a *change rule-based* repair. ASAP-Repair leverages patterns or the change rules to determine edit operations for an AUG representing a misuse and to transform it into a corrected AUG.

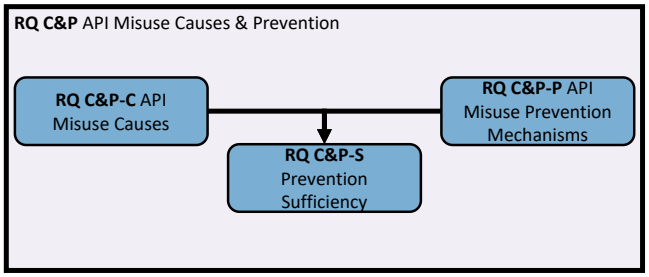
We evaluated ASAP-Repair based on two datasets (i.e., MUBench and AndroidCompass+) first, as an idealized check to demonstrate its principle applicability, and second, as a comparison between the pattern- and rule-based variant. Since ASAP-Repair is an intermediate approach towards a full-fledged API-specific APR technique, we discussed the conceptual differences to existing APR techniques.

8.2. Main Results and Contributions

In this section, we summarize the main results to answer the three main research questions: **RQ C&P** (cf. Section 8.2.1), **RQ D** (cf. Section 8.2.2), and **RQ R** (cf. Section 8.2.3).

Moreover, we present some additional contributions beyond the considered research questions (cf. Section 8.2.4) as well as the big picture of this thesis in Section 8.2.5.

8.2.1. RQ C&P API Misuse Causes & Prevention



RQ C&P-C We found *eleven root causes with 44 different sub-root causes* with their degree of research focus (i.e., number of publications per root cause). For instance, much research has focused on issues with the API documentation or its usability. However, only a few studies have targeted the issues that API developers face in providing useful APIs or problems that client developers encounter in finding the correct resources to learn the API. Moreover, we recognized *interdependencies among these root causes*, a research topic – to the best of our knowledge – not targeted so far. This way, we provide a mean to verbalize hypotheses, named *views*, on these interdependencies. These views serve as a basis for subsequent refinement through empirical studies. Furthermore, we elicited the *typical structure of root cause studies* and determined that a majority of root causes were evaluated with *qualitative studies* offering research abilities towards quantitative studies.

RQ C&P-P We found that prevention mechanisms focus on *recommendations and automated support*, while the research on these mechanisms has some *focus on certain root causes*. Therefore, subsequent research could focus on prevention mechanisms for non-frequently targeted root causes. While studies presenting automated support typically provide an evaluation, *many recommendations lack empirical evidence*. Thus, subsequent research should critically review these recommendations.

RQ C&P-S When mapping prevention mechanisms to root causes, we found that only a few validated recommendations targeted the root causes. Thus, the *research focus was heavily on validated automated support*. Moreover, present *prevention mechanisms did not equally target all identified root causes*. Thus, further research should target the under-analyzed root causes to better understand and tailor prevention mechanisms.

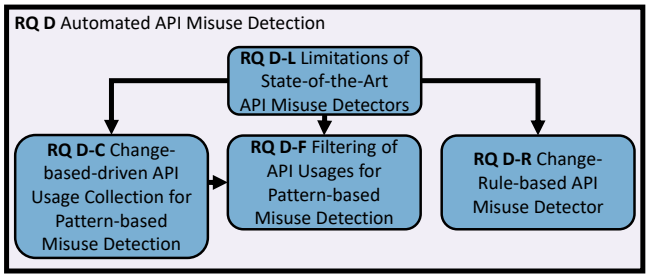
Answering RQ C&P Our results provide – to the best of our knowledge – *the most comprehensive overview of the root causes and prevention mechanisms compared to state-of-the-art surveys*, particularly due to the mapping to research methodologies. We provide all data and scripts via a *replication package*[1]. This way, researchers in software engineering profit from our results and can replicate and build on them. Practitioners benefit from this overview by identifying those prevention mechanisms that have been proven effective in practice. We answer the main **RQ C&P** as follows:

[1] http://doi.org/10.5281/zenodo.15594600

RQ C&P Are root causes of API misuses sufficiently targeted by API misuses prevention mechanisms?

While there exist many prevention mechanisms, the current state of research has three main issues: (1) Some root causes have gained too little research focus and thus require additional empirical studies to obtain knowledge on how to prevent them. (2) The interdependencies among root causes are not sufficiently analyzed, which can impact the effectiveness of prevention mechanisms when applying mechanisms to single root causes. (3) Research on prevention mechanisms is skewed towards single root causes as well as towards automated support, while scientific evidence for recommendations, which could be smoothly added to a developer process, is missing. Therefore, we denote that prevention mechanisms require further research efforts toward these three issues.

8.2.2. RQ D Automated API Misuse Detection



RQ D-L We found various issues and limitations with state-of-the-art API misuse detectors. First, researchers validated them with *different benchmarks*, and the *majority of detectors lack a sufficient replication package if any is available at all.* Second, they are *limited in the collection methodology for donor code* to support a practical scenario as well as *computationally expensive inference techniques.* Therefore, many techniques suffer from *low precision* (i.e., large false positive rate) and, thus, low practical applicability.

RQ D-C A first notion to tackle these issues is to *improve donor code quality* by deliberately embedding the code sample search into the development process. We demonstrated that commits from a Version Control System (VCS) have the potential to support this idea, and we evaluated the effect of our lightweight artifact *RAIX to extract and find relevant code.*

RQ D-F In a further experiment, we validated a set of filter techniques to improve donor code quality. In detail, we found contradicting results to Amann et al. [Ama18, ANN+19b] that *internally and externally obtained source code improves pattern quality.* An interesting result was that *information on the misused API had a negligible effect* on finding donor code. This way, we concluded that RAIX can sufficiently find matching code samples based on the context of the API change. Finally, we suggested *applying the more fine-grained method- rather than file-filtering to exclude non-useful code examples.* Even though not statistically significant, we observed a small positive effect on pattern-based misuse detection.

RQ D-R We determined that our *change rule inference successfully creates change rules for real-world API misuse and its fixes* using ChaRLI. We tested various components of our misuse detector RuDetect based on these change rules and found the largest precision by

using *a threshold-based applicability check* and a *vector similarity metric based on the graph vectorization named Exas vectors* [NNP+09a]. This way, we demonstrated *significantly better precision than the pattern-based misuse detector MUDetect*. A *crucial factor for a comparable recall, however, is the training data*, namely, those fixes from which change rules are inferred. Thus, further research should explore possibilities to improve training data for RuDetect.

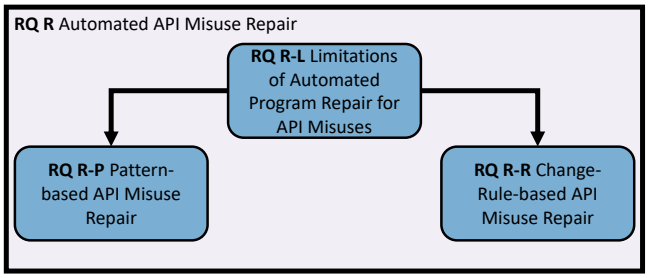
Answering RQ D We provided a *comprehensive and updated overview of the research conducted in API misuse detection* and demonstrated their limitations. We targeted these limitations with the common idea of *leveraging change information from VCS*. The first variant *improves pattern inference* by finding more related donor code based on an API usage change. The second variant derives *change rules from previous fixes* and obtains better results than conventional pattern-based misuse detection.

Thus, our results provide evidence for software engineering research that change information support pattern-based detection, and it enables a better misuse detection based on previous fixes. Moreover, we provide different datasets, a replication package, and implemented prototypes for replication and reuse. Practitioners benefit from the overview of previous misuse detection results to decide on potential misuse detectors as well as on a process to include pattern-based misuse detection into their development process (e.g., a Continuous Integration (CI)-system). Thus, we answer **RQ D** as follows:

RQ D How can we improve the precision of state-of-the-art API misuse detectors within a realistic software development process?

We found and demonstrated a positive effect compared to state-of-the-art misuse detectors, with two techniques leveraging change information from the VCS. (1) RAIX and subsequent search and filter strategies improving the pattern inference for pattern-based misuse detection. (2) RuDetect with integrated ChaRLI leveraging previous fixes outperforming previous pattern-based misuse detectors.

8.2.3. RQ R Automated API Misuse Repair



RQ R-L We found that state-of-the-art APR techniques have no sufficiently large benchmarks for API misuse repair, particularly when using Deep Learning (DL) techniques. Moreover, we identified that many techniques require automated tests for fault localization which is an issue for the successful repair of API misuses [KMSH21]. Additionally, we found that some techniques have non-sufficient patch mechanisms for API misuses or too restrictive assumptions on the type of API.

RQ R-P We contributed the pattern-based variant of ASAP-Repair, which does not require tests for fault localization, but its concept applies state-of-the-art pattern-based misuse detectors. We demonstrated that it is, in principle, applicable to obtaining 75.9% of

the patches in the idealized check. However, we found limitations when applying it in a more realistic scenario (i.e., no valid patches), mainly due to ineffective pattern inference.

RQ R-R In a similar way, the rule-based variant of ASAP-Repair does not require tests for fault localization as well. We demonstrated its applicability in an idealized check (i.e., 62.9% valid patches) as well as in the more realistic scenario (i.e., 102 valid patches, from which our misuse detector RuDetect detected 91). Thus, it outperforms the pattern-based variant. Even though it is limited to repairing AUGs, it is a strong counterpart to state-of-the-art APR for API misuses.

Answering RQ R We presented an overview of the limitations of APR techniques towards API misuses. Due to these limitations, we suggested ASAP-Repair, an API-specific APR technique. In contrast to the majority of state-of-the-art APR, ASAP-Repair does not require automated tests for fault localization by leveraging patterns and change rules used in previous API misuse detection. We demonstrated the principle applicability of both variants, while rule-based repair obtains more valid fixes in a realistic scenario. We provide ASAP-Repair together with the data and analysis scripts as a replication package[1].

Currently, ASAP-Repair is an intermediate step towards a full-fledged API-specific APR technique. Still, software engineering researchers benefit from our insights and artifacts to explore new APR techniques. While these results are somewhat limited for practitioners, repaired AUGs can serve as a template to manually derive valid fixes for API misuses. Thus, we answer **RQ R** as follows:

RQ R Does API-specific information support automated API misuse repair?

We found that current APR techniques did not fully facilitate API misuse repair. By leveraging API-specific information in the form of API usage patterns and change rules from API misuse detection, we found promising results for improved API misuse repair. These results serve as the basis for full-fledged API-specific APR techniques.

8.2.4. Additional Results

Next to the three main research questions, we made some further contributions and observations, which are interesting for the software engineering research community.

New API Misuse Datasets While some datasets with API usages and misuses exist (cf. Section 5.2.2) [ANN+16, KL21], these lack some generalizability due to their collection process and contain some natural bias, for instance, many similar API misuses. We target this issue by providing *AndroidCompass* [NBKO21b] as well as its extension *AndroidCompass+* (cf. Section 6.4.1), a dataset of Android compatibility checks. While in this thesis, AndroidCompass+ serves as an API misuse dataset, it is also applicable for research studies on API evolution and Android compatibility checks, in particular. Moreover, we also shipped our crawling script, allowing the construction of updated or similar datasets.

Targeting Flaws in the Pattern Mining Process Pattern-based misuse detection suffers from high false positive rates. One issue lies in how samples from, which patterns are inferred, are collected. For better transparency from which code parts instances of pattern

originate, we developed an IDE plugin named *SpecTackle* [HNO19] to support API pattern mining researchers. This plugin also allows the comparison of multiple different pattern mining techniques and data structures.

Non-Reusable Replication Packages We found that the majority of misuse detectors have no available replication packages (56.7% cf. **Insight D-5** on page 107), and many available replication packages are hardly reusable. Interestingly, Zhang et al. [ZFM+23] found in the domain of learning-based APR that 16 out of 44 techniques (≈ 36.4%) with a linked repository lack source code, a related dataset, or a trained model.

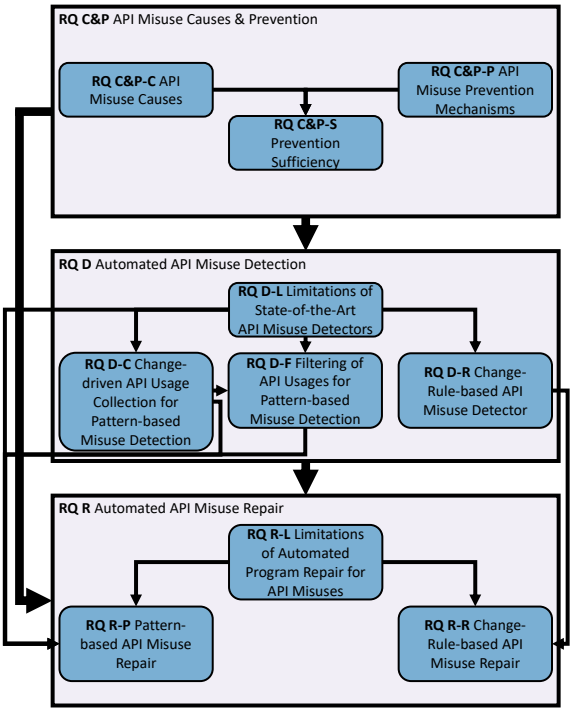
In general, we and others [HNKO20, HWS20, THLH21, WTH+22, BEH+23] explored this issue for the whole software engineering research domain and found non-availability and non-reusability of software artifacts as a prevalent issue. This issue impedes the replicability of research results not only for API misuses but software engineering as a whole domain.

8.2.5. Conclusive Results

We targeted the problem of API misuses separated into its root causes & prevention mechanisms as well as techniques for detection and repair.

We found that research on root causes and their prevention is not sufficient, and thus, new techniques for detection and repair are required. Moreover, the analysis of root causes also improves the body of knowledge on API misuse, helping to foster better localization techniques (e.g., finding misuse in resources such as Q&A pages).

Moreover, we showed a strong link between API misuse detection and its repair by reusing patterns and change rules as repair templates. Due to the similar idea to leverage change information from VCS, these techniques have the potential for embedding into a practical CI process.



We formulate the conclusive result as follows:

API Misuses: Causes, Prevention, Detection, and Repair

- *We provide data that there exist a variety of API misuse causes & prevention mechanisms, which, however, require further research;*
- *We present techniques to lift up state-of-the-art API misuse detectors to practical use cases as well as a new technique for more precise API misuse detection;*
- *We demonstrate the ability with a prototypical technique that reuses information from misuse detection to automatically repair API misuses.*

8.3. Further Research

Further Studies on Root Causes & Prevention Mechanisms for API Misuses We discussed in **RQ C&P** the issues on the research of several root causes, which have been rarely analyzed or could be approved or falsified by other research methodologies. Similarly, we motivate more empirical studies on processes and recommendations for root cause prevention. The implementation of small process steps could be more convenient and acceptable than automated techniques whose results could be mistrusted (e.g., due to high false positive rates).

Extended API Usage Patterns The thesis only considered intraprocedural patterns representing API usages in single method declarations in client code. Upcoming studies should determine the prevalence of interprocedural usages and misuses as well as the degree of interprocedure distribution, for instance, among methods in a single class. Existing research does not cover this sufficiently, such as by Zhong et al. [ZM19], who only analyzed API usage at an intraprocedural level.

Regarding techniques to infer interprocedural patterns, a simple variant is to inline method declarations and infer AUGs. Other techniques may expand AUGs towards a system AUG similar to a system dependence graph. However, a statically obtained API usage representation may become too complex and overapproximate the behavior (i.e., it represents relationships among API elements that are not possible at execution). In contrast, dynamically obtained representations require valid input data to represent the full behavior.

Moreover, we used locally available libraries, meaning we did not involve the misuse of Web-APIs such as RESTfull APIs. Thus, upcoming research should analyze whether existing misuse detectors apply to such APIs as well. Another extension denotes patterns in which multiple APIs interact with each other [HNO18].

Fair Comparison to Full-Fledged API-Specific APR We already discussed in Section 7.5.3 potential extensions of ASAP-Repair for a full-fledged repair technique, namely, one editing the source code. Moreover, even though we find a suitable way to apply ASAP-Repair on source code, further research has to provide a fair experimental comparison to other APR techniques.

A possible basis denotes the APIARTy framework by Kechagia et al. [KMSH21]. However, currently, this framework relies on a small API misuse dataset and thus should be extended with a larger dataset, for instance, misuses from AU500 [KL21] or our AndroidCompass+ dataset (cf. Section 6.4.1).

Another issue of this framework is that correct behavior is checked via automated tests. Automated tests require a runnable version of the program to repair, which can cause issues, for instance, if certain dependencies for building the source code are not available anymore. Moreover, tests do not represent the full spectrum of the source code behavior. Therefore, we suggest that subsequent research should provide ground truth for those API misuses, for instance, from the real programmers' solutions. This way, validation of APR techniques could be simplified by directly comparing the generated patch to the real-world source code (e.g., by using code clone detection). Another benefit is that techniques using tests for fault localization and patch validation do not suffer from the overfitting phenomena of solutions, namely, solutions that only satisfy the given test cases but do not generalize [LTLLG18, PMK+24].

Another important feature is the training data for techniques inferring patches from previous usages. Particularly, in light of advances in machine and deep learning [ZFM+23, RRWF25], it is important to have a common training dataset among different techniques. This training data could be constructed to contain potential solutions but also avoid issues such as data leakage (i.e., data containing the ground truth) or temporal bias (i.e., fixing code that was not available at the time of the misuse). These extensions could greatly improve the replicability and comparability of API-specific APR techniques.

In addition, we also suggest a distinction between fault localization and patch generation to clarify whether an APR fails due to a false localization or its inability to generate a patch.

Empirical Studies for Automated Support We require empirical evidence on whether the application of RAIX within a CI process is useful. For instance, if a live code search for donor code lasts too long, an API misuse check would block the CI. In this case, misuse detection may heuristically apply RAIX on certain suspicious commits [NHO18], for instance, on certain branches or changes provided by inexperienced developers.

Similarly, having a full-fledged ASAP-Repair technique, its embedding into a practical use case should be validated through empirical studies. As seen with existing APR techniques applied in practice [KBK+17, NHL18, BSPC19, MBC+19, KWM+21], patches are typically monitored and manually assessed before adding them to the productive code base. Thus, subsequent work should validate these processes in practice using a full-fledged version of ASAP-Repair.

Impact of Machine and Deep Learning There have undoubtedly been tremendous advances in the research and application of machine learning. Large Language Models (LLMs), such as GPT (i.e., **G**enerative **P**re-trained **T**ransformer), gained much attraction, and research successfully applied these models also for various software engineering and, particularly, API-related tasks [FGH+23, ZFM+23, ZWX+23, YHWH24, WHH+24, NMH+24].

Nevertheless, we also observe several issues, such as hallucinating effects [HYM+25] impacting the correctness of solutions. Hidden data leakage and data bias have implications on the reliability of scientific results [ZFM+23]. Privacy, copyright, and ethical issues cause practical and legal hurdles complicating their application [ZXL+24]. Other issues refer to their efficient application regarding energy consumption impacting economical as well as environmental costs in the light of the climate change [SDSE20, SGM20]. Therefore, we suggest that machine learning should not be considered a 'silver bullet' to solve all tasks around API misuses, particularly their root causes, prevention, detection, and repair. It should rather be considered as a further technique that has to be critically validated and balanced along the multiple costs and benefits it provides as well as in its context-specific application.

We shortly discuss three possible applications (among many others) using machine learning for API misuses that we believe have great potential:

1. Particularly, LLMs are suitable to *generate human-readable documentation* customized for client developers and, thus, can be applied for automated generation and update of API documentation [FGH+23, NMH+24]. This way, they target the commonly discussed root cause of outdated API documentation and thus can avoid the false application of deprecated APIs.

2. DL can be used to obtain *API embeddings*, such as by Nguyen et al. [NNPN17], who demonstrated this with the API2Vec embedding. This can target the issue of missing or rarely available API usage examples due to the ability to relate API usage from multiple different programming languages. This way, we can target the issue of insufficient training data, which then can improve the recall of pattern-based misuse detection, our techniques RuDetect, and APR based on our technique ASAP-Repair [NHKO20b]

3. *Metric learning* is a data-driven technique to derive a similarity metric [BHS15]. While our metric used for RuDetect is deduced from the graph vectorization based on Exas vectors, we do not claim that it performs best. This way, we assume a trained metric derived by metric learning can better discriminate correct usages, their misuses, and completely different usages (e.g., different API or different context). Thus, we expect improvements for RuDetect.

Appendix

A.1. Appendix API Misuse Causes & Prevention

A.1.1. Discussion Detailed API Misuse Root Causes

Regarding the most frequently found detailed root causes, the following causes with 10 or publications were identified:

- *(I-1): missing/insufficient documentation* (20)
- *(J-1): inconsistencies due to Application Programming Interface (API) changes* (17)
- API is designed *(A-1): too complex* (16) and/or *(A-2): too abstract* (13)
- client developers have *(C-2): missing domain knowledge* (14)
- *(E-5): API incompatibilities* (11)
- *(E-4): ambiguous/unclear usage* (10)

A.1.2. API Misuse Root Causes Examples from Literature

For better illustration, we present a set of examples from the analyzed literature for the detailed root causes in Table A.1.

Table A.1.: Determined codes of API misuse root causes with examples from the literature

	general root causes		**detailed root causes**	**example from the literature**
(A)	complexity and abstraction issues	(A-1)	too complex	structure of the API with hard to identify correct sequences [NKMB16]

continued subsequently

Table A.1.: Determined codes of API misuse root causes with examples from the literature (continued)

	general root cause		**detailed root cause**	**example from the literature**
(A)	complexity and abstraction issues	(A-2)	too abstract	lack of understanding the implementation/constraints [NKMB16]
		(A-3)	compromise design issues	trade-off between "make hard things possible" and "keep simple things easy" [ZHKG20]
(B)	human API developer issues	(B-1)	communication issues between API and client developer	lack of understanding the mental models of client developers by API developers [ZHKG20]
		(B-2)	unclear usage scenarios	unknown use cases of an API and no templates [ZHKG20]
		(B-3)	missing education on API design	only learning API design in practice [MKA+18]
		(B-4)	heterogene API client users	larger geographical distribution of APIs increases error proneness [CS14]
(C)	human client developer issues	(C-1)	developer process-related issues	time constraints and interruptions during learning [Rob09]
		(C-2)	missing domain knowledge	conceptional misunderstanding of security concepts [NDT+17]
		(C-3)	non-helping developer experience	no correlation between experience and detecting security API blindspots [OLR+18]
		(C-4)	mindeset issues	open personality correlated with better detection of API blindspots [OLR+18]
(D)	API code issues	(D-1)	naming issues	irritating naming in comparison to domain knowledge [ZHKG20]
		(D-2)	errors in API	not handling unknown scenarios in the API [Afo15]
		(D-3)	unclear API error/ warning messages	unclear error messages in cryptographic APIs [LYY+23]
		(D-4)	insufficient defaults	insufficient defaults for cryptographic algorithms [LYY+23]
		(D-5)	insufficient initialization of objects	problems when creating objects without constructors [DER12]

continued subsequently

Table A.1.: Determined codes of API misuse root causes with examples from the literature (continued)

	general root cause		detailed root cause	example from the literature
(D)	API code issues	(D-6)	insufficient error handling	not properly reporting the error status [DER12]
(E)	API usability issues	(E-1)	too many features	many features that are not used from a library [QLL16]
		(E-2)	unknown entry points	missing code examples cause lack of entry points [MSS18]
		(E-3)	API customization issues	problems to reuse outdated code examples [KB23]
		(E-4)	ambiguous/unclear usage	ambiguous parameter types [PFM13]
		(E-5)	API incompatibilities	unknown relation between data types [PFM13]
		(E-6)	inconsistent usage compared to similar APIs	contradictions to known conventions [GPT12]
		(E-7)	unknown constraints	unknown limitations of the API [ARB20]
(F)	false API usage resource	(F-1)	outdated APIs	large proportion of code marked as deprecated [QLL16]
		(F-2)	using internal API	using experimental APIs [BSvdB15]
		(F-3)	auto-generation issues	non-trustful code for cryptographic APIs [GALIF20]
		(F-4)	using insufficient API samples	code clones of too complex API examples [NHM+19]
(G)	finding features issues	(G-1)	findability issues of present API features	unknown framework support for certain functionality [NDT+17]
		(G-2)	missing API features	unsupported use cases in the security domain [ABF+17]
(H)	API installation issues	(H-1)	API configuration issues	multiple possibly conflicting configurations [MNY+18]
		(H-2)	technical environment issues	unclear usage environment of the API [KMS14]
(I)	documentation issues	(I-1)	missing/insufficient documentation issues	negative effect on novices of an API without documentation [GPT12]
		(I-2)	too verbose documentation	overwhelming information of the API [TCK21]

continued subsequently

Table A.1.: Determined codes of API misuse root causes with examples from the literature (continued)

	general root cause		detailed root cause	example from the literature
(I)	documentation issues	(I-3)	effort to create documentation	discussed large effort to create and maintain a documentation [BCM22]
		(I-4)	not using documentation	not using available information when asking questions in Q&A platforms [PHR19]
		(I-5)	issues with examples in the documentation	observed negative effect without examples of REST API [SMAR17]
(J)	API evolution issues and breaking changes	(J-1)	inconsistencies due to API changes	breaking API changes in Android ecosystem [MRK13]
		(J-2)	effort supporting old APIs	allowing too many previous versions causing installation issues [CZLF19]
		(J-3)	non-documented API changes	significant amount of not documented deprecation of APIs [WLLC20]
(K)	other context-related issues	(K-1)	domain	domains with higher ratio of false answers and misuses, namely databases, cryptographic, IO, and network [ZUR+18]
		(K-2)	licensing issues	possible restrictions to use an API due to a license [MSS18]
		(K-3)	API corporate development	correlation of failure proneness in corporate projects [CS14]
		(K-4)	issues due to programming language	boilerplate API due to programming langauge constraints [NHM+19]

A.1.3. API Misuse Root Causes Study Methodologies

We present samples of the research methodologies applied when analyzing API misuse root causes in Table A.2.

Table A.2.: Determined codes of API misuse root causes methodologies with examples from the literature

	general method		detailed method	example from the literature
(A)	Raw Resource Collection	(A-1)	library/client code search and project selection	using open-source projects [GWL+19]
		(A-2)	API/client developer recruitment	API developers recruited via personal contacts [ZHKG20]
		(A-3)	guideline and other	analysis of evolution policies of Web APIs [EZG15]
		(A-4)	newsgroup/forum discussion	analysis of StackOverflow posts [ZUR+18]
		(A-5)	issue-tracking/ pull requests	analysis of **C**ommon **V**ulnerabilites and **E**xposures (**CVE**s) [ZER11]
		(A-6)	version control data	commit analysis [ANBL18]
(B)	Preparation of Resources and Studies	(B-1)	sampling	random sampling of bugs [GWL+19]
		(B-2)	keyword-based search	scraping and filtering Q&A posts [MNY+18]
		(B-3)	developer study task creation	self-constructed programming tasks [SM08]
		(B-4)	developer study group splitting	using a control group without software artifact [DH09b]
(C)	Code-related Studies	(C-1)	dynamic code analysis	dynamic Android developer tools [CZLF19]
		(C-2)	static code analysis	linking Android classes to Q&A questions [LVBDP+14]
		(C-3)	engineering research	automated detection of code constraints and whether they are documented [SSD15]
		(C-4)	API misuse detection	applying API misuse detector on collected bugs [GWL+19]
		(C-5)	mining error stacks	analysis of stack trace data from Android [KMS14]
		(C-6)	manual code analysis	analysis for usability smells [LYY+23]
(D)	Developer-related Studies	(D-1)	pre-study	previous survey regarding security-related problems [ABF+16]
		(D-2)	case study	applied guideline on a own API [MRARMB+18]

continued subsequently

Table A.2.: Determined codes of API misuse root causes methodologies with examples from the literature (continued)

	general method		**detailed method**	**example from the literature**
(D)	Developer-related Studies	(D-3)	exploratory study	extending a learning model to understand the learning processes of APIs [GVIK20]
		(D-4)	developer study	programming study tasks using APIs [DER12]
		(D-5)	survey	questionnaire on API learning [MSS18]
		(D-6)	interview	mailing interview with API developers on reasons for breaking APIs [BXHV18]
		(D-7)	longitudinal observation study	long observation of Android app incompatibilities [CZLF19]
(E)	Other Studies/ Analysis	(E-1)	meta analysis	excluded
		(E-2)	theory inference	theory inference on robust API knowledge [TCK21]
		(E-3)	literature review	literature review on API usability measurement [SK15]
		(E-4)	document analysis	n-gram analysis of Q&A posts [LS20]
(F)	Study-related Processes	(F-1)	recording and transcription	observation and personal notes during development task [SC07]
		(F-2)	iterative process	internal followed by an external qualitative analysis [Afo15]
		(F-3)	think-aloud technique	think-aloud of client developers during experimental programming task [SM08]
(G)	Obtaining Results	(G-1)	manual	manual analysis of Q&A posts [GWL+19]
		(G-2)	expert assessment	API usability experts validating code based on a guideline [GPT12]
		(G-3)	measuring cognitive load/dimension	using a cognitive load questionnaire after experimental programming task [WA19]
		(G-4)	measuring usability	applied self-developed API usability measurement framework [SK15]

continued subsequently

Table A.2.: Determined codes of API misuse root causes methodologies with examples from the literature (continued)

	general method		detailed method	example from the literature
(G)	Obtaining Results	(G-5)	topic/taxonomy extraction	extending API misuse taxonomy [HLXX23]
		(G-6)	(open/closed) coding	open coding of transcribed developer interviews on Web API documentation [BCM22]
		(G-7)	other measurements	measuring scores on task [NDT+17]
		(G-8)	descriptive analysis/ statistical tests	analysis of survey results on API learning using mathematical statistic tests [RD11]

A.1.4. API Misuse Root Causes Mapping

We present a mapping of detailed root causes to detailed methodologies in Figure A.1.

A.1.5. API Misuse Prevention Mapping

We present a mapping of detailed root causes to different classes of prevention mechanisms in Figure A.2. Moreover, Table A.3 depicts a qualitative overview of different prevention mechanisms targeting the general root causes.

Figure A.1.: Correlation of detailed root causes and applied research methodologies

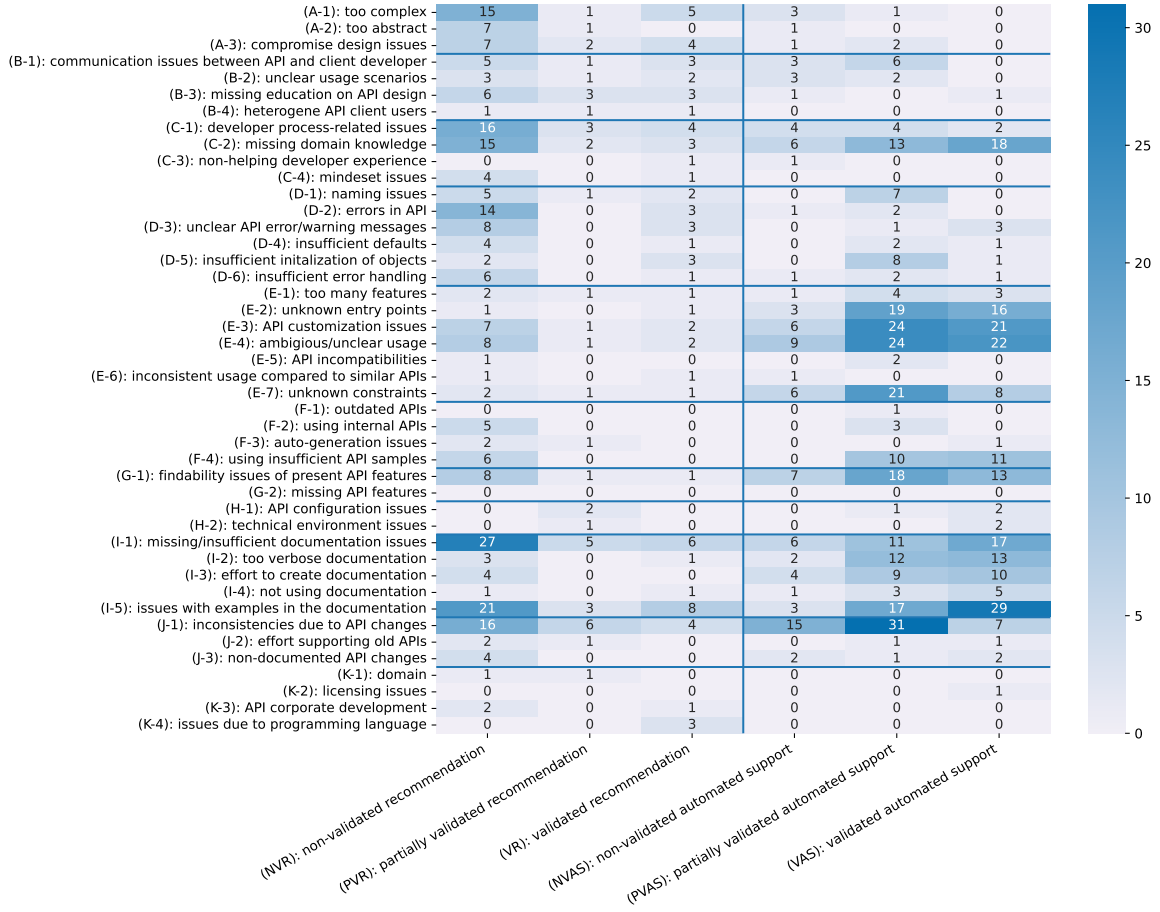


Figure A.2.: Correlation of detailed root causes and prevention mechanisms

Table A.3.: Summary of prevention mechanisms based on the qualitative analysis with most relevant literature

Misuse Cause	Type	Detailed Prevention Mechanism	References
(A) *complexity and abstraction issues*	VR	adhering to API design principles (e.g., easy-to-learn)	[ZWH19]
		simplifying common client tasks	[FMGK24]
		improving error handling	[GALIF20, FMGK24]
		simplified test modes	[IKND16]
		secure defaults	[IKND16, FMGK24]
		early integration for API test	[MRARMB+18]
(B) *human API developer issues*	VR	review process with client developers	[WBK21]
		test processes of API usability	[FWZ10]
		feedback to up-to-dateness with repository badges	[TZKV18]
	VAS	support adherence to functional requirements and design decisions	[SY21]
(C) *human client developer issues*	VR	no copy&paste for secure code	[ABF+17]
		priming of security requirements	[NDT+17, DNRS21]
		strict usage constraints with few exemptions	[WBK21]
		client developer training with examples and pair programming	[SAKW21]
	VAS	support of the selection from Q&A pages	[FXK+19, WJZ+23, CPY+24]
		automated in-code suggestions and generation	[IHK17, KRW+23, MWD24, NMH+24]
		API and library recommendation	[LZP+23]
		find expert client developers in issue trackers	[SWT+21]
		support of finding and sharing common knowledge	[LLS+18, HLH+22]
		support of improved warnings	[NWA+17, GIW+18]
(D) *API code issues*	VR	avoid linguistic anti-patterns and unusual terms	[ANBL18, MRARMB+18]
		code resilience against internal errors	[MRARMB+18]
		adhering to API design principles	[ZWH19, FMGK24]
		SCA with more precise messages	[TVBW21]
		constructors instead of factories	[ESM07]

continued subsequently

Table A.3.: Summary of prevention mechanisms based on the qualitative analysis with most relevant literature (continued)

Misuse Cause	Type	Detailed Prevention Mechanism	References
(D) *API code issues* (cont.)	VR	reporting build status in library repositories	[TZKV18]
	VAS	automated security advices and warnings	[NWA+17, GIW+18, LAH18]
(E) *API usability issues*	VR	no binding of API methods to helper classes	[SM08]
		strict usage constraints with few exemptions	[WBK21]
		using examples from unit tests	[NM10]
		priming for security task	[DNRS21]
		constructors instead of factories	[ESM07]
	VAS	find related or alternative code examples or API elements	[HWM06, BDWK10, DR11, KAS13, IHK17, NKZ17, ZUR+18, LHT+22]
		summarizing code examples and discussion from Q&A platforms	[HAHH15, GZHK18, NGB21]
		mining idiomatic code examples, constraints, and patterns	[MXBK05, DH09a, SHA14, BSZ+20, LPM+21, NDRDS+22]
		automated code generation	[KAB20, SWZ+20, MWD24, YHWH24]
		customize online code examples	[ZYLK19, TS21]
		enhancing documentation with client data	[SFYM09, SXC+19]
		improve discoverability of API element with annotations	[SM17]
		automated summary generation of API classes	[LPM+19]
(F) *false API usage resources*	VAS	validating and add warnings to Q&A posts	[FXK+19, HWL21, MUR23]
		provide code examples for Q&A posts	[HAHH15, SWZ+20, YHWH24]
		finding and summarizing Q&A posts	[WPWZ19, LHT+22, LCC+23, CPY+24]
		customize online code examples	[ZYLK19, TS21]
		detect frauds of up-voting low quality Q&A posts	[MUKS24]
(G) *finding features issues*	VR	no API method placement in helper classes	[SM08]
	VAS	improve ordinary code search and example suggestions	[EM21, KRW+23, BFH24]

continued subsequently

Table A.3.: Summary of prevention mechanisms based on the qualitative analysis with most relevant literature (continued)

Misuse Cause	Type	Detailed Prevention Mechanism	References
(G) *finding features issues* (cont.)	VAS	API element and library recommendation	[SPK14, HXX+18, CXL19, LZP+23]
		enhancing documentation	[SFYM09, UKR20, NMVH23]
		enhance the customization and summary of Q&A posts	[GZHK18, ZYLK19]
		extending APIs with annotation for better discoverability	[SM17]
(H) *installation issues*	VAS	optimization system as domain specific language for Deep Learning (DL) systems	[WLY+24]
		automated privacy configurations	[SAI20]
		automated dependency issues configuration	[LWL+22]
(I) *API documentation issues*	VR	code examples in the documentation	[WZF+12, SAKW21, BCM22, GMWI22]
		enhance documentation using crowd sourced information	[DPCdAM16, TRJ+21]
		documentation constructed to support different human learners of APIs	[MSS20]
		highlighting potential errors when using the API	[UBv+22]
	VAS	find additional code examples, templates, patterns of API usage	[MXBK05, HWM06, DH09b, BW12, KLHK13, IHK17, NKZ17, GZHK18, HGHH18, ZYLK19, BSZ+20, SXP+21, ZJR+21, WBI+23, BFH24]
		find additional information from crowd-sourced information	[SFYM09, BDWK10, GFX+10, TR16, CYYZ19, WPWZ19, ZJR+21, WJZ+23, BFH24]
		improve search and orientation abilities in the documentation	[RC15, LLS+18, WJZ+21, HLH+22, NMVH23]
		automated documentation generation	[MM16b, UKR20, UKR21, GMBG+23]
		support for tutorials	[HJS+20, GOSN22, NZS+23]

continued subsequently

Table A.3.: Summary of prevention mechanisms based on the qualitative analysis with most relevant literature (continued)

Misuse Cause	Type	Detailed Prevention Mechanism	References
(I) *API documentation issues* (cont.)	VAS	automated detection of inconsistencies between documentation and code	[MLK19, ZWY+20]
		support of writing and updating documentation	[LWCK21, HMM23]
		interactive documentation or integration into the development environment	[DH09a, NR25]
		quality assessment of documentations	[BCH+23]
(J) *API evolution issues and breaking changes*	VR	apply semantic versioning	[RvV17]
	VAS	automated API migration techniques	[SPK14, GAQ+18, LSC22, DND+25]
		incompatibility checks for client applications caused by library upgrades	[NDBB20, ZZW+23]
		automated updates of the documentation	[LWCK21, HMM23]
		suggesting alternative libraries	[CXL19]
		create compatibility code	[DNMJ08]
(K) *other context related issues*	VR	static over dynamic types	[MHR+12, EHRS14, PHR14]
		characteristics of secure development in the domain of smart contracts	[WXL+21]
	VAS	automated fix of license issues in forked projects	[HXC+24]

A.1.6. Detailed Comparison API Misuse Research Effort Causes & Prevention

We present a comparison of the research effort (i.e., rr_{effort}) of general prevention mechanisms to detailed root causes (cf. Figure A.3), of recommendation to detailed root causes (cf. Figure A.4), and of automated support to detailed root causes (cf. Figure A.5).

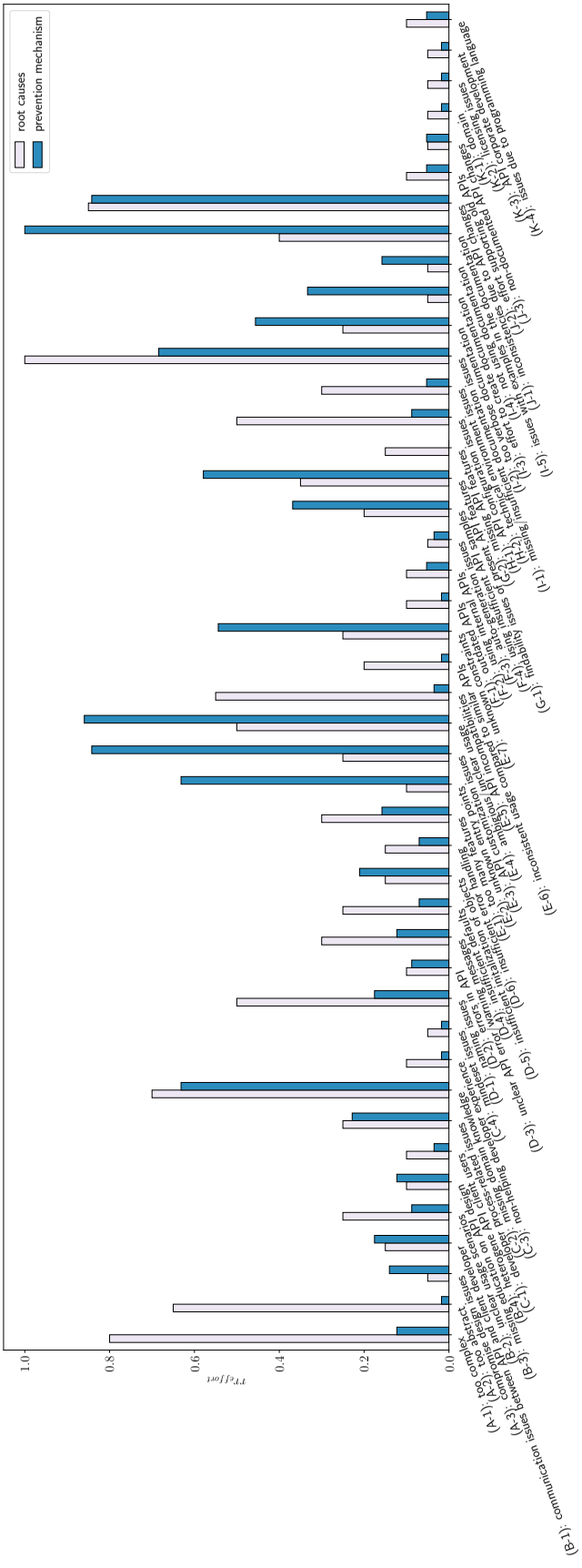


Figure A.3.: Detailed Comparison of relative research effort (rr_{effort}) between research on root causes and prevention mechanisms in general

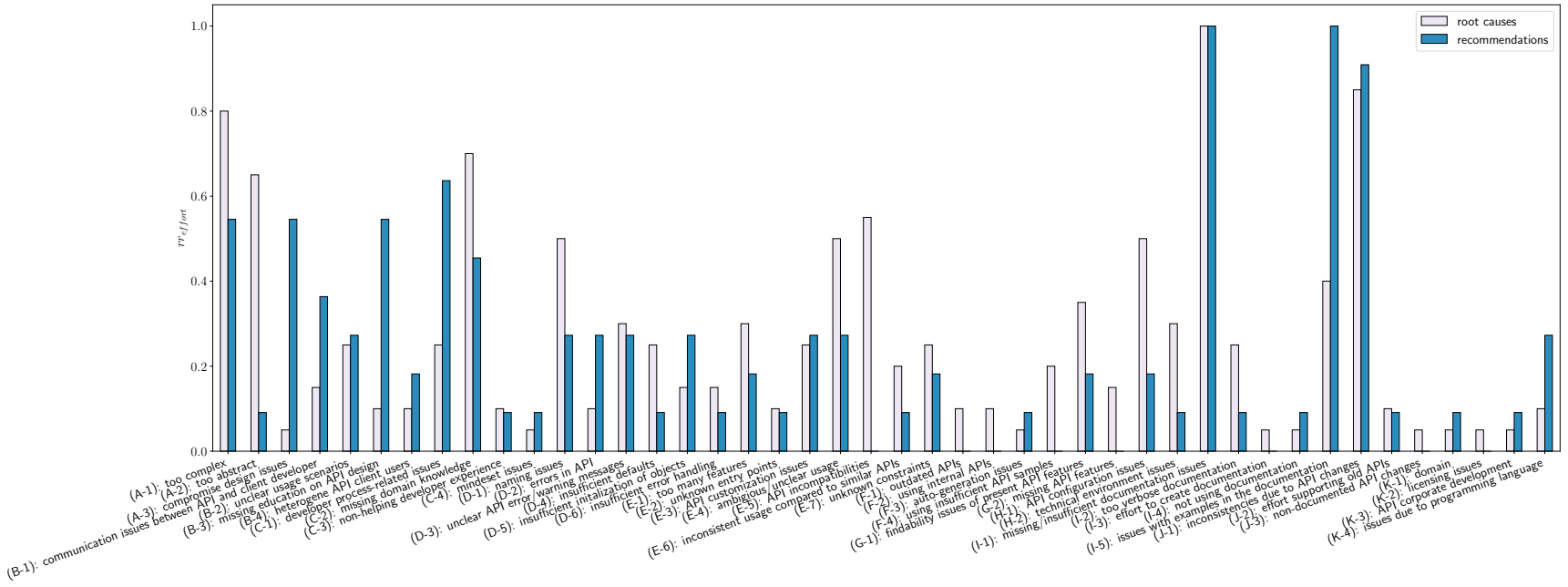


Figure A.4.: Detailed Comparison of relative research effort (rr_{effort}) between research on root causes and recommendation

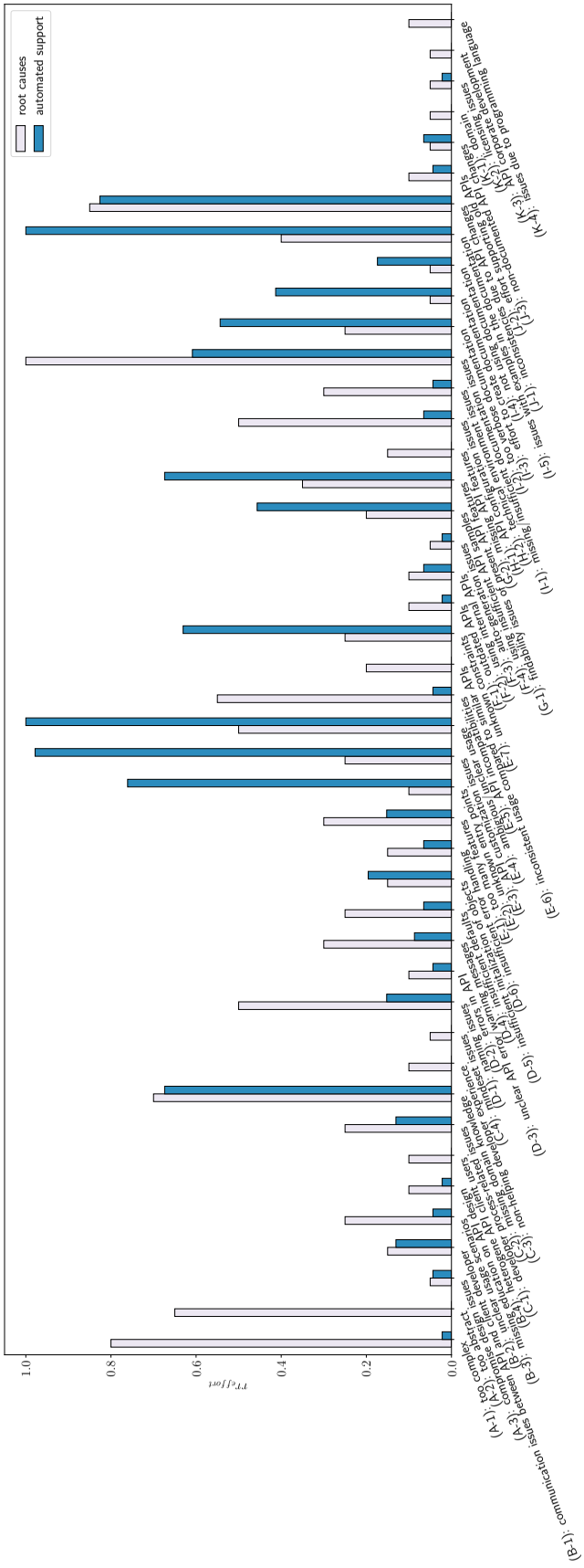


Figure A.5.: Detailed Comparison of relative research effort (rr_{effort}) between research on root causes and automated support

A.2. Appendix Improving Pattern-Based API Misuse Detection

A.2.1. URLs to API Misuse Detectors

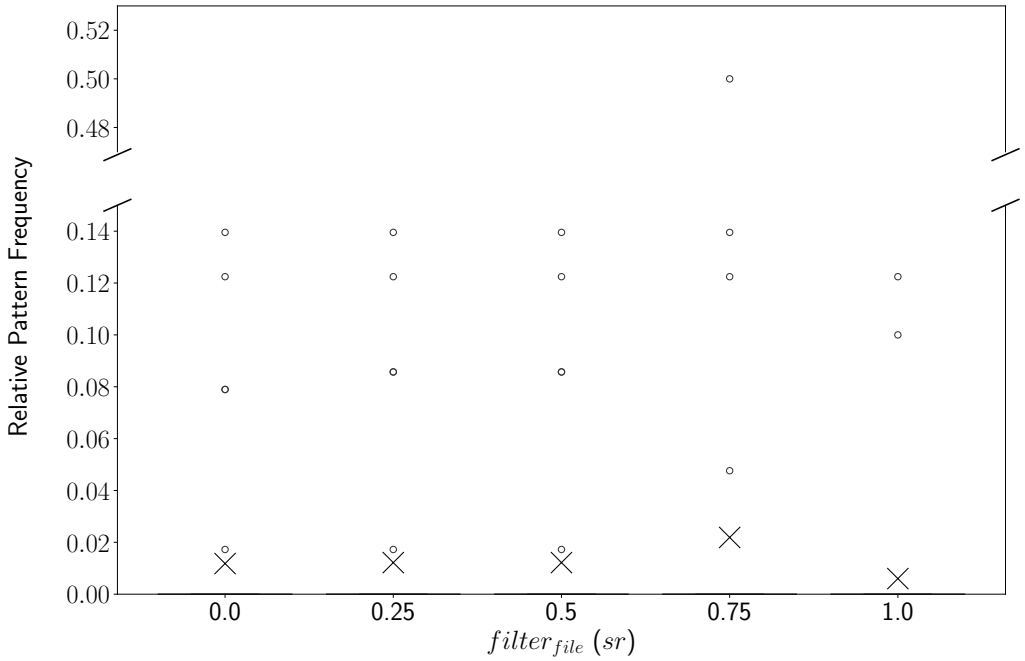
In Table A.4, we provide all found URLs of the replication packages of the analyzed API misuse detectors.

A.2.2. Additional Results Filtering Commits

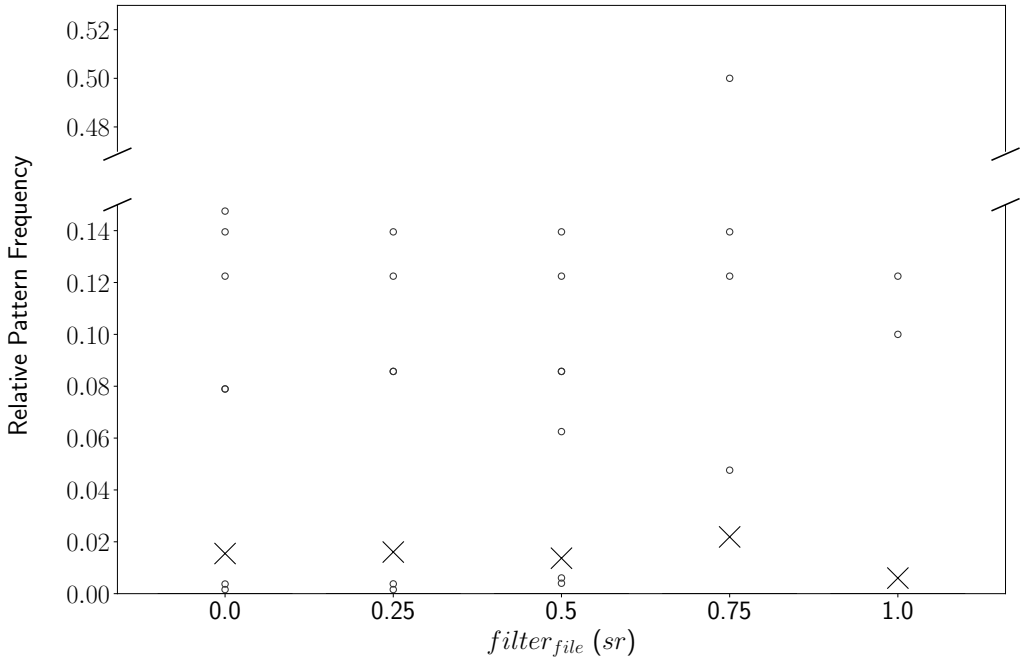
We present further experimental results of filtering strategies tested as improvements in finding the donor code for Frequent Pattern Mining (FPM), namely, API specification mining. In detail, we present different with different *satisfaction ratios (sr)* by using all and only misused API imports from internal (cf. Figure A.6) and external source code (cf. Figure A.7).

Table A.4.: Replication Packages of API isuse Detectors with available detectors highlighted in gray.

	Short Term	URL	Note
SES	APDetect [WXQ23]	-	not available
	APISan [YMS+16]	https://github.com/sslab-gatech/apisan	available - not provided but found on GitHub
	APP-Miner [JWL+24]	https://github.com/JiangJias/APP-Miner	available
	CAR-Miner [TX09b]	http://ase.csc.ncsu.edu/projects/carminer/	URL is not working
	CL-Detector [ZCSZ21]	https://github.com/subZHS/CL-Detector	available but missing merging concept (cf. https://github.com/subZHS/CL-Detector/issues/1)
	CPAM [LCP+21]	https://cpam2019.wixsite.com/mysite	available but only mining framework but missing static detectors
	DMMC [MBM10]	https://www.stg.tu-darmstadt.de/research/core/	URL is not working
	FuzzyCatch [NVN20]	https://bitbucket.org/tamnguyenthe/exassist_repo/src/master/	available
	GrouMiner [NNP+09b]	-	not available
	Jadet [WZL07]	https://www.st.cs.uni-saarland.de/models/jadet/JADET.zip	available
	MUDetect [ANN+19b]	https://github.com/stg-tud/MUDetect	available
	PR-Miner [LZ05]	-	not available
	SpecCheck [NK11]	http://www.comp.nus.edu.sg/~anhcuong/tools/speccheck.html	URL is not working
	Tikanga [WZ11]	https://www.st.cs.uni-saarland.de/models/tikanga/index.php3	linked artifact is not available
MES	Acharya/Xie [AX09]	-	not available
	Alattin [TX09a]	https://sites.google.com/site/asergrp/projects/alattin/	no artifact available
	APICAD [WZ23]	https://github.com/x2018/apicad_public	available
	CrySL [KSA+21]	https://github.com/CROSSINGTUD/CryptoAnalysis https://github.com/CROSSINGTUD/Crypto-API-Rules	available
	Doc2Spec [ZZXM09]	-	not available
	EMDetect [ÇM18]	https://github.com/ervina/emdetect	repository is empty
	Li et al. [LZT+24]	https://doi.org/10.6084/m9.figshare.24552193	available
	OCD [GS10]	-	not available
	Pradel et al. [PJAG12]	-	not available
	Pradel/Gross [PG12]	-	not available
	Ren et al. [RYX+20]	-	not available
IS	ALP [KL21]	https://github.com/ALP-active-miner/ALP	available
	ARBITRAR [LMC+21]	https://github.com/petablox/arbitrar	available
	F-LSTM/S-LSTM [OGKY20]	-	not available
	Salento [MCJ17]	https://github.com/trishullab/salento	available
	StandardTrans/Target-ComTrans [YRW22]	-	not available

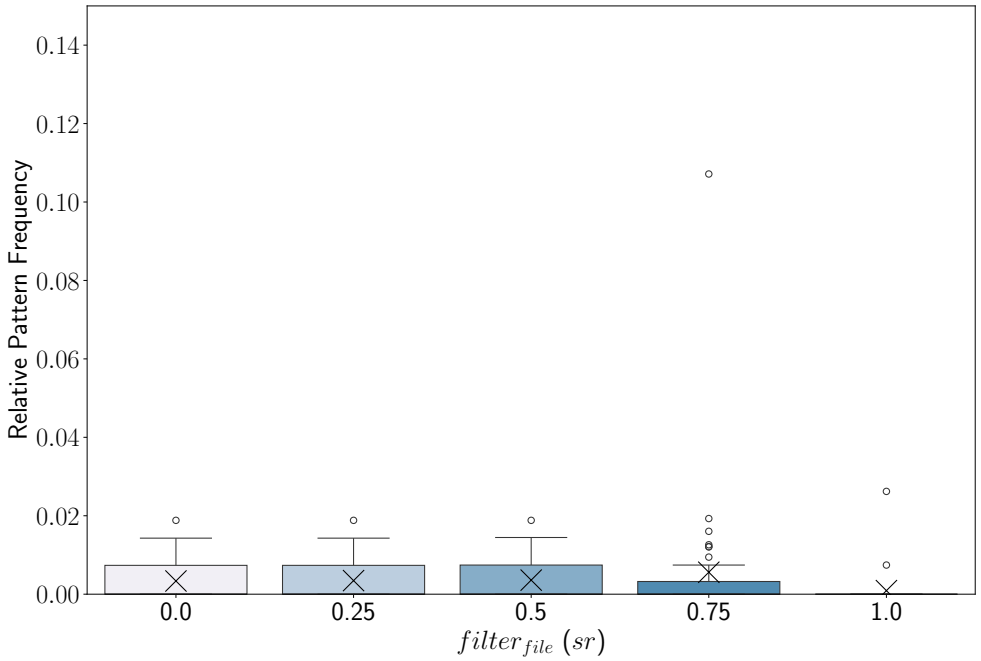


(a) Internal/All Imports

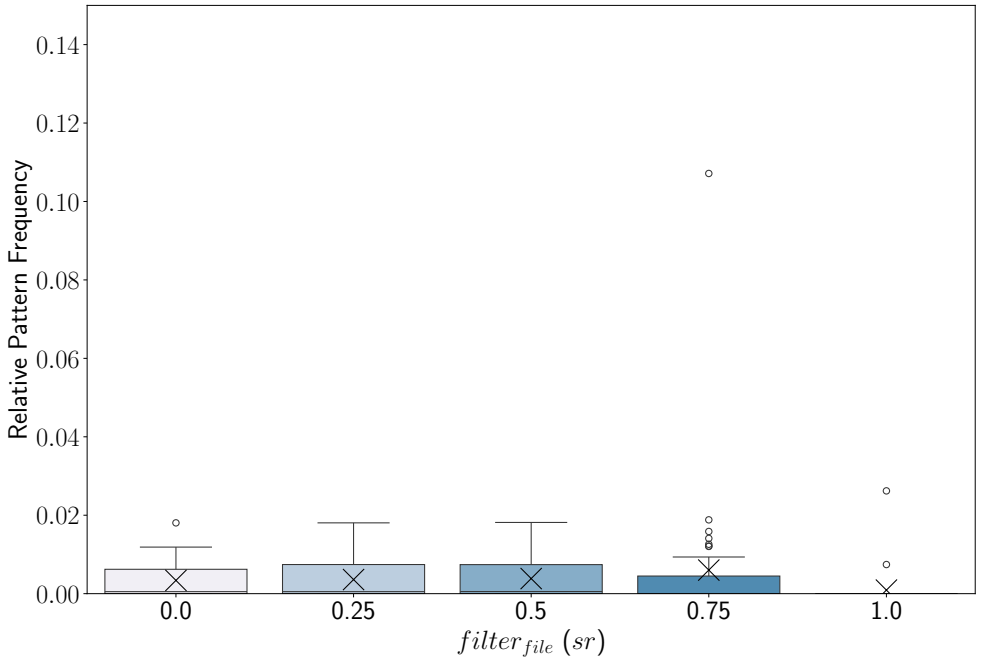


(b) Internal/Misused Imports

Figure A.6.: Distribution of the relative pattern frequency using different file filter strategies grouped by each API search strategy of internally found source code



(a) External/All Imports



(b) External/Misused Imports

Figure A.7.: Distribution of the relative pattern frequency using different file filter strategies grouped by each API search strategy of externally found source code

A.3. Appendix Change Rule-Based API Misuse Detection

A.3.1. Change Rule Inference

Formal Definition of the Cost Function mapping API Usage Graph (AUG) nodes We denote two functions, $V(aug)$ and $E(aug)$, obtaining the set of nodes and edges of an AUG aug, respectively, the node label functions l_{V_m} and l_{V_f} as well as the edge label functions l_{E_m} and l_{E_f} of the respective AUGs aug_m and aug_f. Further, assume a node $n_m \in V(aug_m)$ and a node $n_f \in V(aug_f)$, both of which are the aforementioned partitions. The formal definition of the cost function is as follows:

$$
\begin{aligned}
cost(n_m, n_f) &= cost_{no}(n_m, n_f) + cost_{ed}(n_m, n_f) \\
cost_{no}(n_m, n_f) &= \begin{cases} 0 & \text{if } l_m(n_m) = l_f(n_f) \\ 2 & \text{if } l_m(n_m) \neq l_f(n_f) \wedge (n_m = \epsilon \vee n_f = \epsilon) \\ 4 & \text{else} \end{cases} \\
cost_{ed}(n_m, n_f) &= \min_{\substack{\forall e_m \in E_{in}(n_m, aug_m), \\ \forall e_f \in E_{in}(n_f, aug_f)}} cost_{ed-in}(e_m, e_f) \\
&\quad + \min_{\substack{\forall e_m \in E_{in}(n_m, aug_m), \\ \forall e_f \in E_{in}(n_f, aug_f)}} cost_{ed-out}(e_m, e_f) \\
cost_{ed-in}\left(\begin{matrix} (n_{m1}, n_{m2}), \\ (n_{f1}, n_{f2}) \end{matrix} \right) &= \begin{cases} 0 & \text{if } l_{E_m}((n_{m1}, n_{m2})) = l_{E_f}((n_{f1}, n_{f2})) \wedge \\ & l_{V_m}(n_{m1}) = l_{V_f}(n_{f1}) \\ 1 & \text{else} \end{cases} \\
cost_{ed-out}\left(\begin{matrix} (n_{m1}, n_{m2}), \\ (n_{f1}, n_{f2}) \end{matrix} \right) &= \begin{cases} 0 & \text{if } l_{E_m}((n_{m1}, n_{m2})) = l_{E_f}((n_{f1}, n_{f2})) \wedge \\ & l_{V_m}(n_{m2}) = l_{V_f}(n_{f2}) \\ 1 & \text{else} \end{cases} \\
&\text{with} \\
E_{in}(n, aug) &= \{(n_A, n_B) | (n_A, n_B) \in E(aug) \wedge n_B = n\} \\
E_{out}(n, aug) &= \{(n_A, n_B) | (n_A, n_B) \in E(aug) \wedge n_A = n\}
\end{aligned}
$$

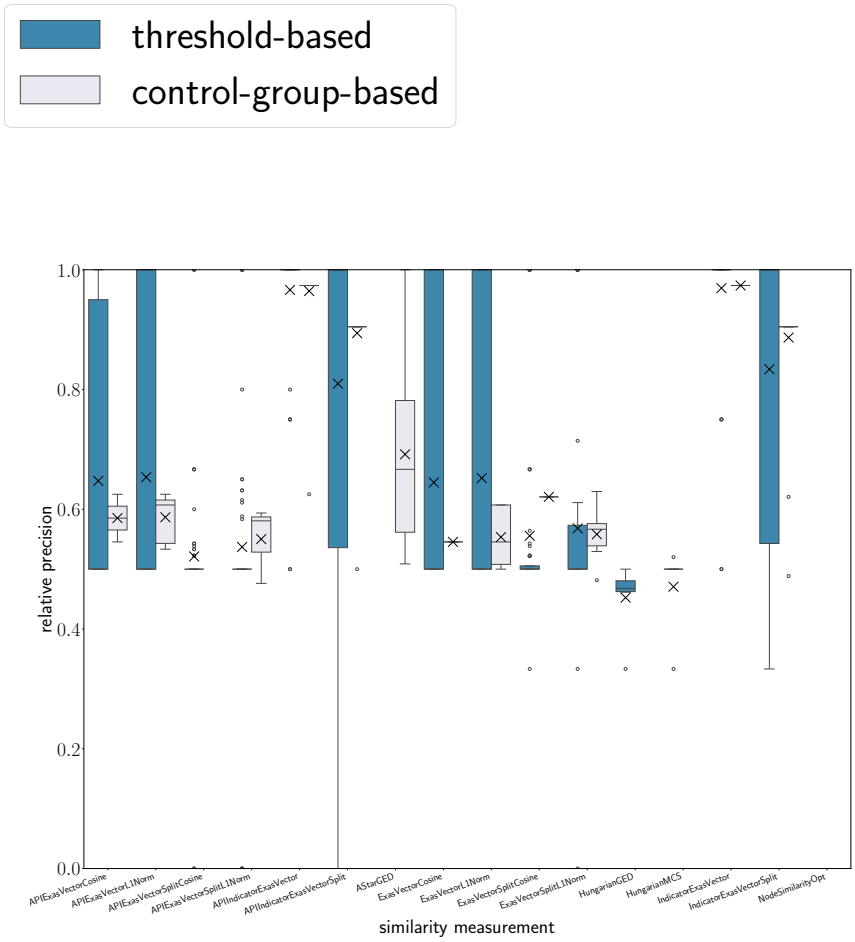
Pseudocode Change Rule Inference (ChaRLI) We present the pseudocode of the change rule generation implemented by ChaRLI based on our previous work [NBKO22] in Algorithm 1. Note that the computation of minimal matching in line 7 is conducted in a first iteration by ignoring differences in `order`-edges the *cost* function. In case this causes an empty change rule, the computation starting in this line and hereafter will be repeated with respecting differences in `order`-edges in the *cost* function.

A.3.2. Detailed Results Applicability Checks

We present a detailed comparison of both applicability checks (i.e., *threshold*-based and control-group-based). In detail, we compare them among all similarity metrics and the relative precision, the conservative precision, and the recall for the MUBench dataset (cf. Figure A.8) and the AU500 dataset (cf. Figure A.9).

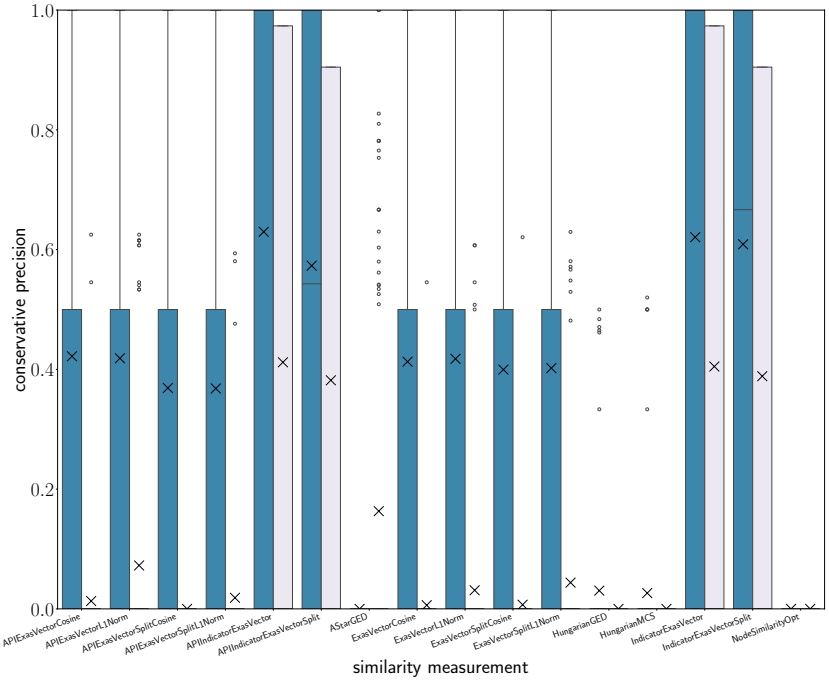
Algorithm 1: Change Rule Generation based on our previous work [NBKO22]

Input: aug_m, aug_f
Result: Change Rule: $(aug_{m'} \rightarrow aug_{f'})$
`/* Create bipartite graph and computing minimal matching. */`
while $|V(aug_m)| \neq |V(aug_f)|$ **do**
  add ϵ-node to AUG with fewer nodes
end
$bipartite \leftarrow new\ WeightedDirectedGraph()$
$bipartite.nodes \leftarrow V(aug_m) \cup V(aug_f)$
$bipartite.edges \leftarrow \{(n_m, n_f, weight : cost(n_m, n_f))|\ n_m \in V(aug_m) \wedge n_f \in V(aug_f)\}$
$minMatch \leftarrow kuhn\_munkres(bipartite)$
`/* Construct basic change rule parts based on minimal matching. */`
$aug_{m'} \leftarrow$ sub graph from aug_m consisting of nodes $n_m \in V(aug_m)$ and their connecting edges, where $costs(n_m, n_f) > 0$ in $minMatch$
$aug_{f'} \leftarrow$ sub graph from aug_f consisting of nodes $n_f \in V(aug_f)$ and their connecting edges, where $costs(n_m, n_f) > 0$ in $minMatch$
`/* Conduct Single Hop Addition. */`
$single\_hop\_node\_pairs \leftarrow$ all action-node pairs represented in $minMatch$ that are connected with an incoming `data-` or `finally`-edge (in aug_m and aug_f, respectively) to one of the nodes in $aug_{m'}$ or $aug_{f'}$
foreach $(sh\_node_m, sh\_node_f) \in single\_hop\_node\_pairs$ **do**
  $aug_{m'} \leftarrow aug_{m'}$ with sh_node_m and its corresponding edges from aug_m
  $aug_{f'} \leftarrow aug_{f'}$ with sh_node_f and its corresponding edges from aug_f
end
$post\_single\_hop\_node\_pairs \leftarrow \emptyset$
foreach $(sh\_node_m, sh\_node_f) \in single\_hop\_node\_pairs$ **do**
  $post\_single\_hop\_node\_pairs \leftarrow post\_single\_hop\_node\_pairs \cup$ all data-node pairs represented in $minMatch$ that are connected with an outgoing `data`-edge (in aug_m and aug_f, respectively) to one of the nodes sh_node_m or sh_node_f
end
foreach $(psh\_node_m, psh\_node_f) \in post\_single\_hop\_node\_pairs$ **do**
  $aug_{m'} \leftarrow aug_{m'}$ with psh_node_m and its corresponding edges from aug_m
  $aug_{f'} \leftarrow aug_{f'}$ with psh_node_f and its corresponding edges from aug_f to $aug_{f'}$
end
`/* Reduce the number of redundant order-edges by applying an adapted version of Hsu's algorithm removing only order-edges. */`
$aug_{m'} \leftarrow hsu(aug_{m'}, \{order\_edge\})$
$aug_{c'} \leftarrow hsu(aug_{c'}, \{order\_edge\})$
`/* Construct final change rule. */`
$(aug_{m'} \rightarrow aug_{f'}) \leftarrow aug_{m'}$ and $aug_{f'}$ connected with `transform`-edges between nodes in $aug_{m'}$ and $aug_{f'}$ based on $minMatch$
return $(aug_{m'} \rightarrow aug_{f'})$

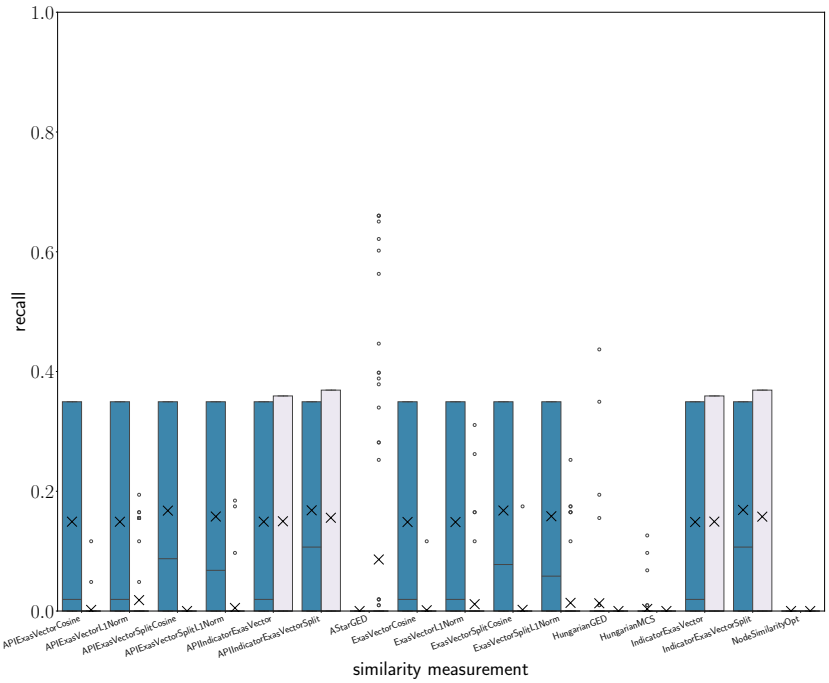


(a) relative precision

Figure A.8.: Comparison the assessment values between Change Rules using *threshold*-based applicability check and control-group-based check on MUBench.

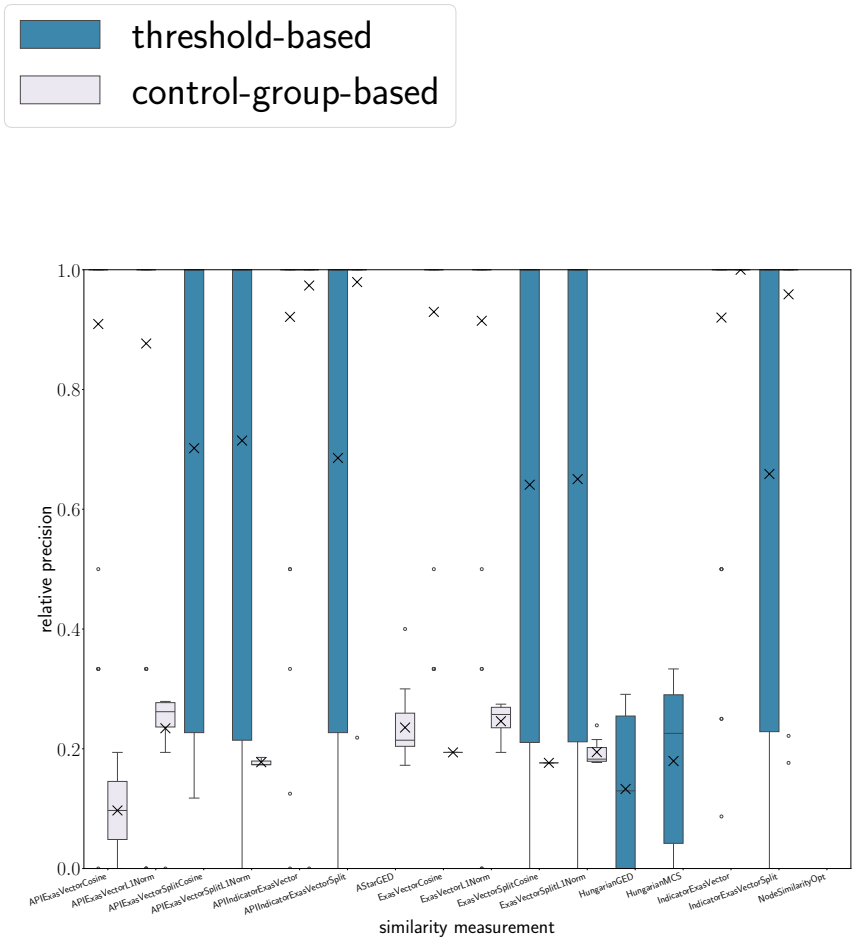


(b) conservative precision



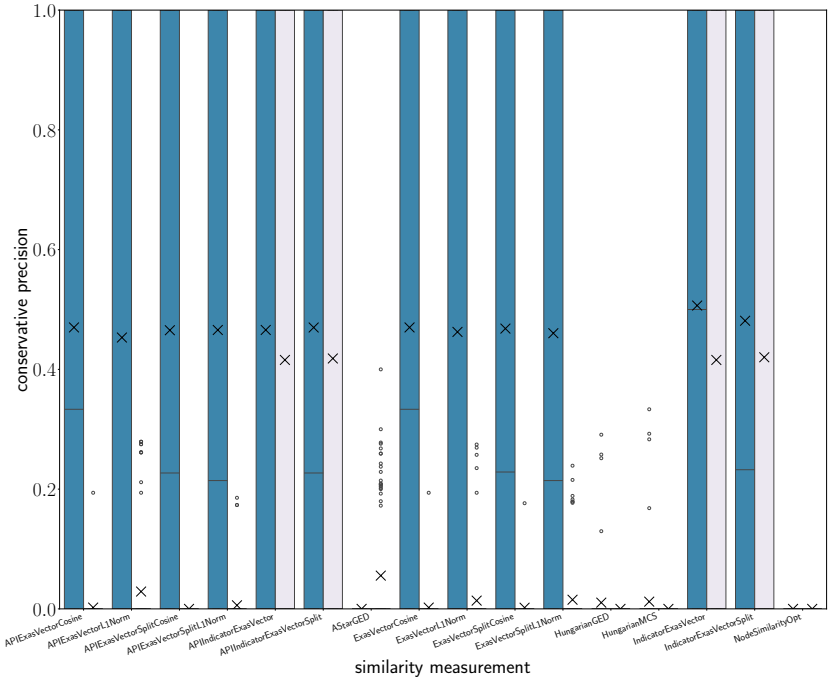
(c) recall

Figure A.8.: Comparison the assessment values between Change Rules using *threshold*-based applicability check and control-group-based check on MUBench (cont.).

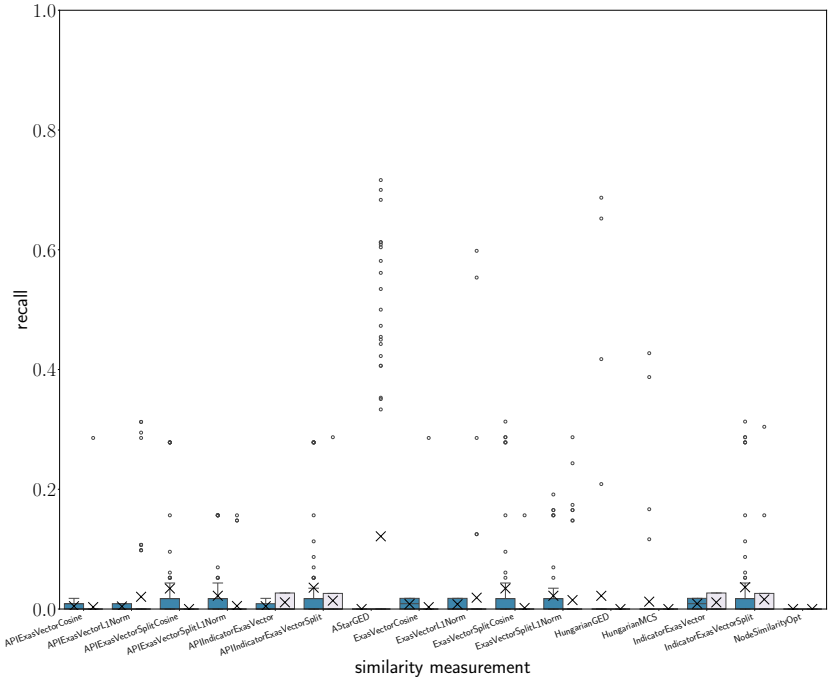


(a) relative precision

Figure A.9.: Comparison the assessment values between Change Rules using *threshold*-based applicability check and control-group-based check on AU500



(b) conservative precision



(c) recall

Figure A.9.: Comparison the assessment values between Change Rules using *threshold*-based applicability check and control- group-based check on AU500 (cont.).

A.4. Detailed Results of RuDetect - Comparison of Similarity Metrics on Misuse Detection

In Figures A.10 and A.11, we depict the differences in the performance in relative and conservative precision as well as recall for both test settings, namely, MUBench-on-MUBench and MUBench-on-AU500 for all 16 similarity metrics. Moreover, we depict the single values in Tables A.5 and A.6.

Further, we provide the differences in the assessment values using change rules *with* and *without* context for the MUBench (cf. Figure A.12 and Table A.7) and the AU500 dataset (cf. Figure A.13 and Table A.8).

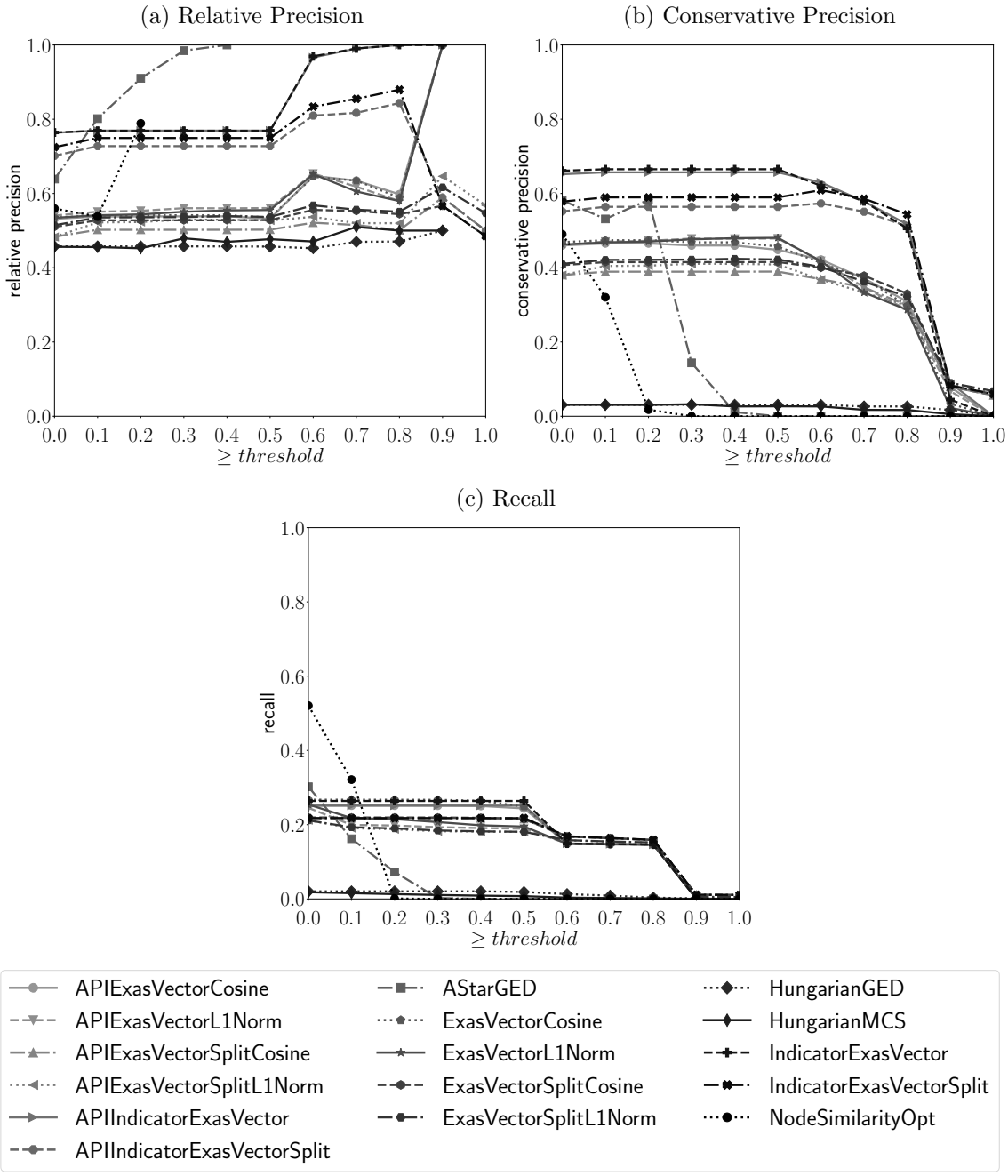


Figure A.10.: Detailed view on the mean assessment values for the MUBench-on-MUBench setting of RuDetect using different similarity measurements and thresholds for the applicability check.

Table A.5.: Mean assessment values for the MUBench-on-MUBench setting of RuDetect using different similarity measurements and thresholds for the applicability check depicted in Figure A.10.

	threshold	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
relative precision	0.0	0.531681	0.540565	0.483476	0.483042	0.763444	0.701397	0.638762	0.535198	0.534527	0.509838	0.514392	0.457487	0.456333	0.764182	0.724572	0.559382
	0.1	0.538031	0.550941	0.502201	0.521955	0.769289	0.727540	0.801649	0.540969	0.541453	0.527727	0.535318	0.457487	0.455710	0.768989	0.749499	0.538044
	0.2	0.538031	0.553261	0.502201	0.523099	0.769289	0.727540	0.910239	0.540966	0.543654	0.527727	0.535852	0.457487	0.452408	0.768989	0.749499	0.788889
	0.3	0.538536	0.560437	0.502204	0.529654	0.769289	0.727540	0.984615	0.541503	0.550781	0.527727	0.536372	0.457487	0.479234	0.768989	0.749499	-
	0.4	0.538494	0.559982	0.502247	0.529266	0.769289	0.727540	1.000000	0.541419	0.554416	0.527867	0.539080	0.457487	0.469774	0.768989	0.749499	-
	0.5	0.531916	0.560780	0.502564	0.527203	0.769289	0.727540	-	0.535457	0.555496	0.528243	0.536861	0.456990	0.476898	0.768989	0.749499	-
	0.6	0.647414	0.653509	0.521211	0.537069	0.966379	0.809760	-	0.644737	0.652047	0.555745	0.568048	0.452408	0.470667	0.969298	0.833783	-
	0.7	0.634615	0.618333	0.515008	0.519977	0.990385	0.817145	-	0.634615	0.605442	0.552736	0.556358	0.469774	0.509804	0.990385	0.854844	-
	0.8	0.597826	0.579545	0.501020	0.519931	1.000000	0.843594	-	0.588889	0.579545	0.544211	0.550958	0.470667	0.500000	1.000000	0.879771	-
	0.9	1.000000	1.000000	0.589080	0.646465	1.000000	0.589080	-	1.000000	1.000000	0.566755	0.616550	0.500000	0.500000	1.000000	0.566755	-
	1.0	-	-	0.500000	0.568182	-	0.500000	-	-	-	0.484848	0.546832	-	-	-	0.484848	-
conservative precision	0.0	0.459993	0.461606	0.380262	0.379920	0.651930	0.551660	0.581345	0.469050	0.462456	0.406725	0.410357	0.030842	0.030764	0.661146	0.578030	0.490245
	0.1	0.465488	0.470467	0.389347	0.404661	0.656921	0.564048	0.531430	0.474108	0.468448	0.415066	0.421037	0.030842	0.030722	0.665305	0.589493	0.320408
	0.2	0.465488	0.472448	0.389347	0.405549	0.656921	0.564048	0.582962	0.474105	0.470352	0.415066	0.421457	0.030842	0.030499	0.665305	0.589493	0.017728
	0.3	0.459873	0.478576	0.389350	0.410631	0.656921	0.564048	0.143820	0.468491	0.476519	0.415066	0.421866	0.030842	0.032308	0.665305	0.589493	0.000000
	0.4	0.459838	0.478187	0.389383	0.410330	0.656921	0.564048	0.011236	0.468419	0.479663	0.415177	0.423995	0.030842	0.026392	0.665305	0.589493	0.000000
	0.5	0.448244	0.478868	0.389629	0.408730	0.656921	0.564048	0.000000	0.457244	0.480597	0.415472	0.422250	0.030808	0.026792	0.665305	0.589493	0.000000
	0.6	0.421910	0.418539	0.368947	0.368104	0.629775	0.573201	0.000000	0.412921	0.417603	0.399637	0.402101	0.030499	0.026442	0.620787	0.608943	0.000000
	0.7	0.370787	0.347378	0.347197	0.333019	0.578652	0.550884	0.000000	0.370787	0.333333	0.378842	0.362571	0.026392	0.017184	0.578652	0.585904	0.000000
	0.8	0.308989	0.286517	0.298360	0.297938	0.516854	0.511843	0.000000	0.297753	0.286517	0.330195	0.321908	0.026442	0.016854	0.505618	0.543679	0.000000
	0.9	0.078652	0.067416	0.079427	0.087164	0.089888	0.079427	0.000000	0.033708	0.022472	0.082784	0.090058	0.016854	0.005618	0.044944	0.082784	0.000000
	1.0	0.000000	0.000000	0.056180	0.063841	0.000000	0.056180	0.000000	0.000000	0.000000	0.059925	0.067586	0.000000	0.000000	0.000000	0.059925	0.000000
recall	0.0	0.251227	0.245337	0.218174	0.211847	0.251445	0.217956	0.302171	0.267372	0.254173	0.219374	0.211520	0.020399	0.018108	0.263990	0.217628	0.520672
	0.1	0.251227	0.199302	0.218174	0.190466	0.251445	0.217956	0.162103	0.267372	0.217956	0.219374	0.192975	0.020399	0.015490	0.263990	0.217628	0.321370
	0.2	0.251227	0.197666	0.218174	0.187520	0.251445	0.217956	0.072652	0.267263	0.214792	0.219374	0.190029	0.020399	0.012981	0.263990	0.217628	0.001200
	0.3	0.251009	0.193302	0.218065	0.182284	0.251445	0.217956	0.002509	0.267045	0.206938	0.219374	0.184357	0.020399	0.010691	0.263990	0.217628	0.000000
	0.4	0.250245	0.190902	0.216974	0.181302	0.251445	0.217956	0.000109	0.264645	0.198320	0.218501	0.181957	0.020399	0.008836	0.263990	0.217628	0.000000
	0.5	0.243700	0.189266	0.215010	0.180430	0.251445	0.217956	0.000000	0.248391	0.194938	0.215010	0.180757	0.018763	0.007527	0.263990	0.217628	0.000000
	0.6	0.149231	0.149122	0.167667	0.157849	0.149231	0.168212	0.000000	0.148686	0.148576	0.167885	0.158285	0.012981	0.003491	0.148686	0.168867	0.000000
	0.7	0.147813	0.147158	0.163085	0.154031	0.147813	0.163303	0.000000	0.147813	0.146722	0.163958	0.154685	0.008836	0.002400	0.147813	0.164067	0.000000
	0.8	0.146286	0.146067	0.158612	0.151740	0.146286	0.158940	0.000000	0.146177	0.146067	0.158940	0.152176	0.003491	0.001527	0.146177	0.159267	0.000000
	0.9	0.000764	0.000655	0.011236	0.005563	0.000873	0.011236	0.000000	0.000327	0.000218	0.011127	0.005454	0.001527	0.000218	0.000436	0.011127	0.000000
	1.0	0.000000	0.000000	0.010581	0.005127	0.000000	0.010581	0.000000	0.000000	0.000000	0.010691	0.005236	0.000000	0.000000	0.000000	0.010691	0.000000

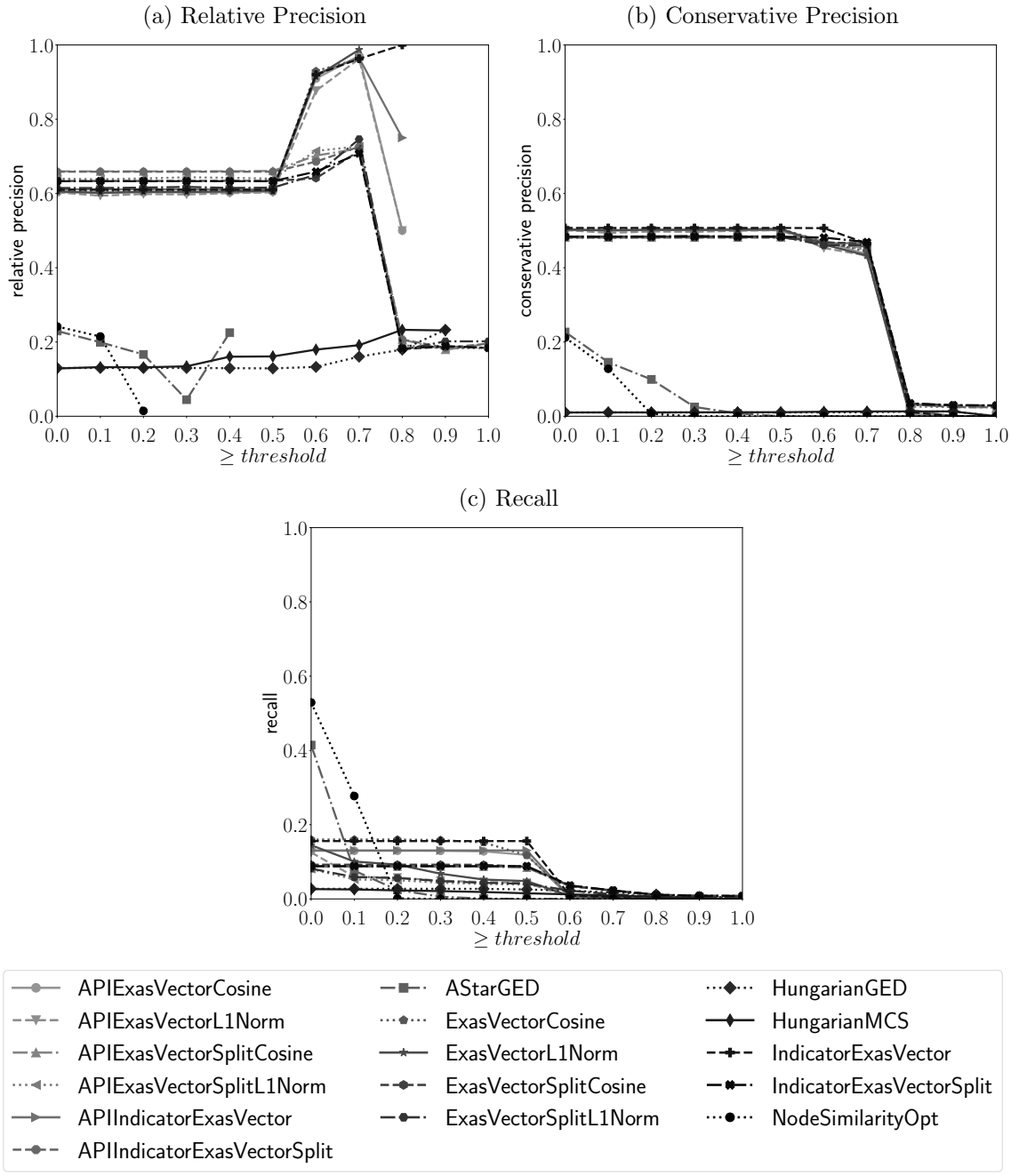


Figure A.11.: Detailed view on the mean assessment values for the MUBench-on-AU500 setting of RuDetect using different similarity measurements and thresholds for the applicability check.

Table A.6.: Mean assessment values for the MUBench-on-AU500 setting of RuDetect using different similarity measurements and thresholds for the applicability check depicted in Figure A.11.

	threshold	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
relative precision	0.0	0.602905	0.603061	0.659614	0.639983	0.611095	0.658634	0.229609	0.606929	0.605730	0.633680	0.614801	0.130009	0.128642	0.610067	0.633078	0.241637
	0.1	0.602905	0.594007	0.659614	0.638226	0.611095	0.658634	0.199231	0.606929	0.601374	0.633680	0.614682	0.130009	0.132367	0.610067	0.633078	0.214802
	0.2	0.602905	0.598483	0.659614	0.640516	0.611095	0.658634	0.166557	0.606967	0.604575	0.633680	0.615847	0.130009	0.131691	0.610067	0.633078	0.014387
	0.3	0.602891	0.597332	0.659719	0.643562	0.611095	0.658634	0.044635	0.606712	0.604028	0.633715	0.617665	0.130009	0.134490	0.610067	0.633078	-
	0.4	0.602945	0.600906	0.659924	0.642253	0.611095	0.658634	0.225071	0.606288	0.605543	0.634124	0.614942	0.129504	0.160424	0.610067	0.633078	-
	0.5	0.604640	0.602914	0.660810	0.639899	0.611095	0.658634	-	0.606963	0.607603	0.633845	0.615760	0.128932	0.161130	0.610067	0.633078	-
	0.6	0.909420	0.876812	0.701984	0.714738	0.921296	0.685641	-	0.929630	0.914815	0.640971	0.650380	0.132852	0.179535	0.920142	0.658886	-
	0.7	0.970833	0.962500	0.721263	0.726813	0.965116	0.726034	-	0.963415	0.987179	0.714429	0.746175	0.160424	0.191464	0.962791	0.707620	-
	0.8	0.500000	0.500000	0.206711	0.191392	0.750000	0.206918	-	-	-	0.180565	0.179143	0.179535	0.232812	1.000000	0.182354	-
	0.9	-	-	0.179160	0.182642	-	0.184619	-	-	-	0.187359	0.201740	0.232812	0.231090	-	0.189088	-
	1.0	-	-	0.195448	0.183711	-	0.195448	-	-	-	0.184030	0.201740	-	-	-	0.184030	-
conservative precision	0.0	0.501291	0.501422	0.481741	0.481785	0.501235	0.481025	0.227029	0.504637	0.503641	0.484160	0.483552	0.010225	0.010118	0.507246	0.483700	0.211772
	0.1	0.501291	0.493894	0.481741	0.480462	0.501235	0.481025	0.145506	0.504637	0.500019	0.484160	0.483458	0.010225	0.010411	0.507246	0.483700	0.127916
	0.2	0.501291	0.497615	0.481741	0.482186	0.501235	0.481025	0.099186	0.504669	0.502681	0.484160	0.484374	0.010225	0.010358	0.507246	0.483700	0.006305
	0.3	0.501280	0.496658	0.481817	0.484479	0.501235	0.481025	0.025076	0.504458	0.502225	0.484187	0.485804	0.010225	0.010578	0.507246	0.483700	0.000000
	0.4	0.501325	0.499630	0.481967	0.483494	0.501235	0.481025	0.007587	0.504104	0.503486	0.484500	0.483663	0.010186	0.010815	0.507246	0.483700	0.000000
	0.5	0.502734	0.501300	0.482614	0.481722	0.501235	0.481025	0.000000	0.504666	0.505198	0.484286	0.484306	0.010141	0.010863	0.507246	0.483700	0.000000
	0.6	0.470037	0.453184	0.465360	0.465784	0.465824	0.469934	0.000000	0.470037	0.462547	0.468125	0.460381	0.010449	0.012103	0.506595	0.481209	0.000000
	0.7	0.436330	0.432584	0.453828	0.449154	0.466292	0.456831	0.000000	0.443820	0.432584	0.465583	0.461119	0.010815	0.012908	0.465169	0.469097	0.000000
	0.8	0.005618	0.005618	0.032516	0.025806	0.016854	0.032549	0.000000	0.000000	0.000000	0.034490	0.030193	0.012103	0.013079	0.011236	0.034832	0.000000
	0.9	0.000000	0.000000	0.024156	0.024626	0.000000	0.024892	0.000000	0.000000	0.000000	0.029472	0.029468	0.013079	0.012983	0.000000	0.029744	0.000000
	1.0	0.000000	0.000000	0.024156	0.024770	0.000000	0.024156	0.000000	0.000000	0.000000	0.028948	0.029468	0.000000	0.000000	0.000000	0.028948	0.000000
recall	0.0	0.129715	0.125201	0.087445	0.080508	0.130618	0.087054	0.414355	0.160012	0.144663	0.092330	0.081681	0.027259	0.026426	0.155799	0.088324	0.528993
	0.1	0.129715	0.059791	0.087445	0.052467	0.130618	0.087054	0.076121	0.160012	0.100722	0.092330	0.058720	0.027259	0.025488	0.155799	0.088324	0.277087
	0.2	0.129715	0.057384	0.087445	0.050318	0.130618	0.087054	0.024053	0.160012	0.092396	0.092330	0.056277	0.027259	0.022986	0.155799	0.088324	0.001605
	0.3	0.129514	0.049157	0.087445	0.043674	0.130618	0.087054	0.005896	0.158608	0.067917	0.092330	0.048070	0.027259	0.021096	0.155799	0.088324	0.000000
	0.4	0.128110	0.045546	0.087250	0.040254	0.130618	0.087054	0.000974	0.151986	0.052167	0.092037	0.043283	0.027064	0.018598	0.155799	0.088324	0.000000
	0.5	0.118479	0.043439	0.084221	0.038398	0.130618	0.087054	0.000000	0.117275	0.048054	0.086468	0.041036	0.025501	0.015148	0.155799	0.088324	0.000000
	0.6	0.004715	0.004314	0.034294	0.022179	0.004715	0.035662	0.000000	0.008327	0.008026	0.034392	0.022179	0.022081	0.012335	0.008929	0.036346	0.000000
	0.7	0.004013	0.003913	0.022472	0.012408	0.004414	0.022960	0.000000	0.004013	0.003913	0.022179	0.012213	0.017587	0.009143	0.004414	0.022863	0.000000
	0.8	0.000100	0.000100	0.010845	0.005862	0.000201	0.011236	0.000000	0.000000	0.000000	0.011041	0.005765	0.011724	0.005637	0.000100	0.011334	0.000000
	0.9	0.000000	0.000000	0.008012	0.004983	0.000000	0.008500	0.000000	0.000000	0.000000	0.008403	0.004885	0.005374	0.001640	0.000000	0.008500	0.000000
	1.0	0.000000	0.000000	0.008012	0.004983	0.000000	0.008012	0.000000	0.000000	0.000000	0.008012	0.004885	0.000000	0.000000	0.000000	0.008012	0.000000

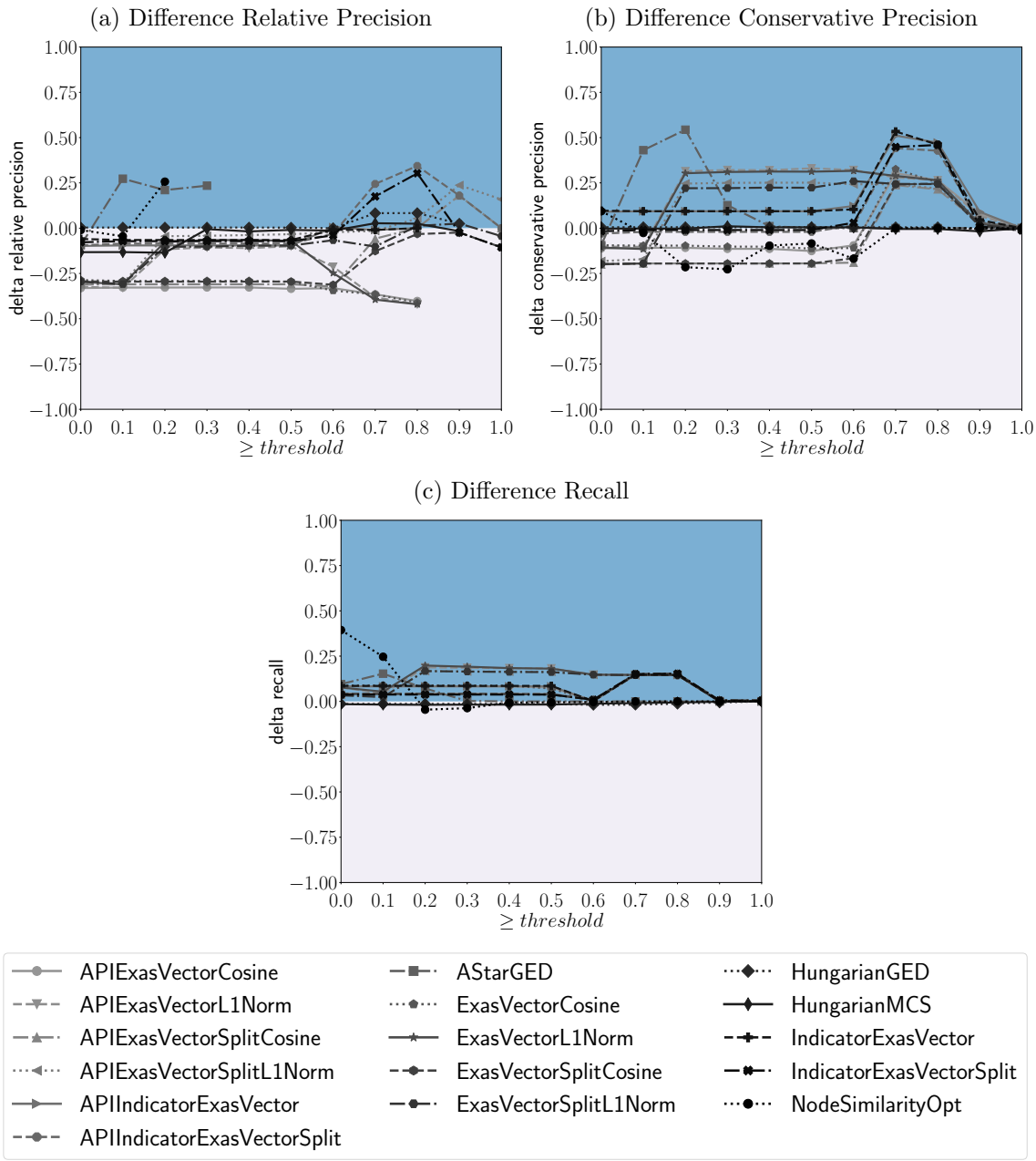


Figure A.12.: Detailed view on the mean assessment values for the MUBench-on-MUBench setting of RuDetect between rules *with context* subtracted by rules *without context* using different similarity measurements and thresholds for the applicability check

Table A.7.: Difference in the Mean assessment values for the MUBench-on-MUBench setting of RuDetect between rules *with context* subtracted by rules *without context* using different similarity measurements and thresholds for the applicability check depicted in Figure A.12.

	threshold	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
relative precision	0.0	-0.330890	-0.324271	-0.311483	-0.310622	-0.098024	-0.094992	-0.093932	-0.286416	-0.296399	-0.293877	-0.294150	0.003306	-0.132497	-0.059743	-0.078052	-0.010393
	0.1	-0.327860	-0.329196	-0.310072	-0.304324	-0.095494	-0.085353	0.272987	-0.291512	-0.310000	-0.295164	-0.308643	0.003306	-0.132114	-0.065323	-0.071701	-0.044749
	0.2	-0.327860	-0.116628	-0.310072	-0.044533	-0.095494	-0.085353	0.210239	-0.291391	-0.076616	-0.295164	-0.102854	0.003306	-0.135725	-0.065323	-0.071701	0.256779
	0.3	-0.327356	-0.106424	-0.310069	-0.042121	-0.095494	-0.085353	0.234615	-0.290855	-0.066178	-0.295164	-0.101977	0.003306	-0.005282	-0.065323	-0.071701	-
	0.4	-0.327345	-0.110358	-0.310026	-0.038032	-0.095494	-0.085353	-	-0.290814	-0.065498	-0.295024	-0.096889	0.003316	-0.017837	-0.065323	-0.071701	-
	0.5	-0.334831	-0.099743	-0.309709	-0.033429	-0.095494	-0.085353	-	-0.296614	-0.069974	-0.294648	-0.096131	0.002819	-0.009558	-0.065323	-0.071701	-
	0.6	-0.331310	-0.213158	-0.322496	-0.022514	0.005386	-0.031122	-	-0.344625	-0.247953	-0.312621	-0.066536	-0.000176	-0.013679	-0.009425	-0.039640	-
	0.7	-0.365385	-0.381667	-0.057948	-0.015023	-0.009615	0.244189	-	-0.365385	-0.394558	-0.127278	-0.101454	0.082712	0.027191	-0.009615	0.174830	-
	0.8	-0.402174	-0.420455	0.001020	0.053264	0.000000	0.343594	-	-0.411111	-0.420455	-0.032712	0.012496	0.083190	0.024612	0.000000	0.302848	-
	0.9	-	-	0.179990	0.237374	-	0.179990	-	-	-	-0.024154	0.025641	0.024612	-0.016667	-	-0.024154	-
	1.0	-	-	0.000000	0.159091	-	0.000000	-	-	-	-0.106061	-0.044077	-	-	-	-0.106061	-
conservative precision	0.0	-0.111823	-0.101995	-0.200326	-0.181887	0.090523	-0.029972	-0.093721	-0.094079	-0.107055	-0.198319	-0.198320	0.005326	-0.002316	0.096434	-0.017175	0.093323
	0.1	-0.108530	-0.103106	-0.194760	-0.170948	0.093355	-0.020505	0.430450	-0.096469	-0.115132	-0.195167	-0.195339	0.005326	-0.002302	0.093473	-0.010259	-0.026649
	0.2	-0.108530	0.314384	-0.194760	0.246101	0.093355	-0.020505	0.543636	-0.096387	0.303088	-0.195167	0.220515	0.005326	-0.002542	0.093473	-0.010259	-0.215444
	0.3	-0.114145	0.321226	-0.194757	0.250020	0.093355	-0.020505	0.126966	-0.102001	0.310147	-0.195167	0.221037	0.005326	0.010532	0.093473	-0.010259	-0.227198
	0.4	-0.114146	0.320017	-0.194724	0.250976	0.093355	-0.020505	0.011236	-0.101988	0.312496	-0.195057	0.223915	0.005327	0.004477	0.093473	-0.010259	-0.096308
	0.5	-0.126341	0.330436	-0.194478	0.251249	0.093355	-0.020505	0.000000	-0.113051	0.311931	-0.194762	0.223107	0.005293	0.004929	0.093473	-0.010259	-0.084270
	0.6	-0.094944	0.321161	-0.190364	0.242355	0.122285	0.015762	0.000000	-0.109551	0.316479	-0.166265	0.259498	0.005073	0.004674	0.103933	0.029932	-0.168539
	0.7	0.303371	0.279963	0.237756	0.242851	0.511236	0.441443	0.000000	0.325843	0.288390	0.241311	0.244312	0.004647	-0.004506	0.533708	0.448373	0.000000
	0.8	0.264045	0.264045	0.214090	0.219286	0.471910	0.427574	0.000000	0.252809	0.264045	0.245926	0.243256	0.004674	-0.004512	0.460674	0.459409	0.000000
	0.9	0.078652	0.067416	0.028865	0.036602	0.089888	0.028865	0.000000	0.033708	0.022472	0.009751	0.017024	-0.004512	-0.017603	0.044944	0.009751	0.000000
	1.0	0.000000	0.000000	0.005618	0.013279	0.000000	0.005618	0.000000	0.000000	0.000000	-0.013109	-0.005448	0.000000	0.000000	0.000000	-0.013109	0.000000
recall	0.0	0.083342	0.077779	0.043308	0.036326	0.083342	0.041453	0.095451	0.088688	0.076906	0.041235	0.033272	-0.013745	-0.015381	0.085742	0.037635	0.394022
	0.1	0.083342	0.042762	0.043308	0.024108	0.083342	0.041453	0.153158	0.088688	0.053671	0.041235	0.024108	-0.013745	-0.017563	0.085742	0.037635	0.246646
	0.2	0.083342	0.188066	0.043308	0.167558	0.083342	0.041453	0.071670	0.088688	0.197884	0.041235	0.167448	-0.013745	-0.018763	0.085742	0.037635	-0.045817
	0.3	0.083124	0.185230	0.043198	0.165049	0.083342	0.041453	0.002182	0.088470	0.191557	0.041235	0.164612	-0.013745	-0.017999	0.085742	0.037635	-0.037199
	0.4	0.082470	0.183593	0.042108	0.164394	0.083342	0.041453	0.000109	0.086288	0.183921	0.040471	0.162758	-0.013527	-0.017890	0.085742	0.037635	-0.006545
	0.5	0.076906	0.182393	0.040362	0.163630	0.083342	0.041453	0.000000	0.071125	0.180866	0.037308	0.161667	-0.015163	-0.017345	0.085742	0.037635	-0.004909
	0.6	0.002509	0.147922	0.008291	0.147158	0.002400	0.008618	0.000000	0.002073	0.147486	0.008400	0.147267	-0.018763	-0.012763	0.001854	0.009163	-0.003273
	0.7	0.147158	0.146504	0.150213	0.144322	0.147158	0.150431	0.000000	0.147376	0.146286	0.150758	0.144649	-0.017890	-0.009381	0.147376	0.150867	0.000000
	0.8	0.145849	0.145849	0.152394	0.147486	0.145849	0.152722	0.000000	0.145740	0.145849	0.152722	0.147922	-0.012763	-0.005782	0.145740	0.153049	0.000000
	0.9	0.000764	0.000655	0.005345	0.001636	0.000873	0.005345	0.000000	0.000327	0.000218	0.005018	0.001309	-0.005782	-0.001964	0.000436	0.005018	0.000000
	1.0	0.000000	0.000000	0.004691	0.001200	0.000000	0.004691	0.000000	0.000000	0.000000	0.004582	0.001091	0.000000	0.000000	0.000000	0.004582	0.000000

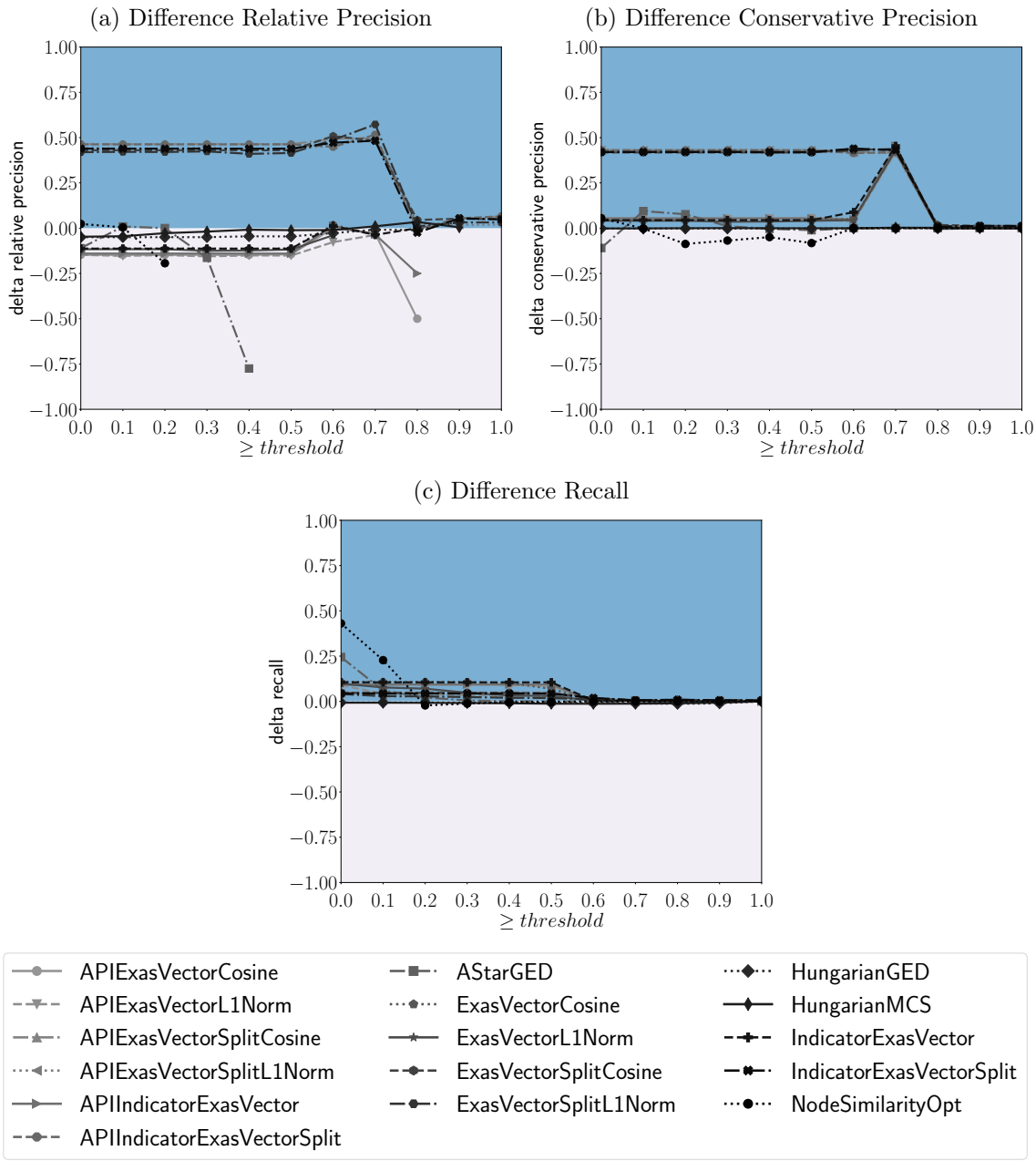


Figure A.13.: Detailed view on the mean assessment values for the MUBench-on-AU500 setting of RuDetect between rules *with context* subtracted by rules *without context* using different similarity measurements and thresholds for the applicability check.

Table A.8.: Difference in the mean assessment values for the MUBench-on-AU500 setting of RuDetect between rules *with context* subtracted by rules *without context* using different similarity measurements and thresholds for the applicability check depicted in Figure A.13.

	threshold	APIExasVectorCosine	APIExasVectorL1Norm	APIExasVectorSplitCosine	APIExasVectorSplitL1Norm	APIIndicatorExasVector	APIIndicatorExasVectorSplit	AStarGED	ExasVectorCosine	ExasVectorL1Norm	ExasVectorSplitCosine	ExasVectorSplitL1Norm	HungarianGED	HungarianMCS	IndicatorExasVector	IndicatorExasVectorSplit	NodeSimilarityOpt
relative precision	0.0	-0.147476	-0.147555	0.463522	0.443604	-0.140017	0.462697	-0.110727	-0.113013	-0.116259	0.430254	0.419334	-0.049763	-0.047371	-0.112988	0.439009	0.023407
	0.1	-0.147476	-0.153361	0.463522	0.439692	-0.140017	0.462697	0.009160	-0.113013	-0.116163	0.430254	0.421332	-0.049763	-0.044552	-0.112988	0.439009	0.004338
	0.2	-0.147476	-0.148571	0.463522	0.438419	-0.140017	0.462697	0.000221	-0.112975	-0.116180	0.430254	0.420451	-0.049763	-0.027775	-0.112988	0.439009	-0.193703
	0.3	-0.147516	-0.156098	0.463627	0.445389	-0.140017	0.462697	-0.163698	-0.112857	-0.124966	0.430289	0.423730	-0.049763	-0.019000	-0.112988	0.439009	-
	0.4	-0.147710	-0.150917	0.463492	0.433021	-0.140017	0.462697	-0.774929	-0.113794	-0.123423	0.430400	0.410217	-0.044889	-0.008396	-0.112988	0.439009	-
	0.5	-0.144146	-0.151366	0.463784	0.435847	-0.140017	0.462697	-	-0.112252	-0.118043	0.429509	0.414990	-0.045536	-0.011198	-0.112988	0.439009	-
	0.6	-0.000302	-0.077035	0.493458	0.458712	0.013160	0.448563	-	0.019569	-0.039031	0.508607	0.486961	-0.027289	-0.010369	0.012006	0.471539	-
	0.7	-0.029167	-0.037500	0.497376	0.511241	-0.034884	0.516140	-	-0.036585	-	0.482218	0.572799	-0.008396	0.010305	-0.037209	0.482953	-
	0.8	-0.500000	-	-0.005743	0.027106	-0.250000	-0.005536	-	-	-	0.045500	0.009143	-0.010369	0.034314	0.000000	-0.024789	-
	0.9	-	-	0.049290	0.018357	-	0.054749	-	-	-	0.052294	0.031740	0.034314	0.004775	-	0.054023	-
	1.0	-	-	0.065578	0.019425	-	0.065578	-	-	-	0.048965	0.031740	-	-	-	0.048965	-
conservative precision	0.0	0.054436	0.054426	0.431065	0.431036	0.053944	0.430390	-0.109483	0.043551	0.041243	0.417875	0.419860	-0.001894	-0.001748	0.044167	0.420464	0.054842
	0.1	0.054436	0.048832	0.431065	0.429156	0.053944	0.430390	0.094251	0.043551	0.040473	0.417875	0.420456	-0.001894	-0.001516	0.044167	0.420464	-0.002146
	0.2	0.054436	0.052741	0.431065	0.429959	0.053944	0.430390	0.076758	0.043583	0.041074	0.417875	0.420706	-0.001894	-0.000393	0.044167	0.420464	-0.087219
	0.3	0.054409	0.047986	0.431142	0.433266	0.053944	0.430390	0.011031	0.043610	0.035342	0.417902	0.422611	-0.001894	0.000230	0.044167	0.420464	-0.068436
	0.4	0.054306	0.051915	0.431204	0.429423	0.053944	0.430390	-0.003649	0.042928	0.036620	0.418117	0.416954	-0.001571	-0.000566	0.044167	0.420464	-0.049358
	0.5	0.056828	0.052121	0.431697	0.428989	0.053944	0.430390	-0.011236	0.044045	0.040458	0.417705	0.418886	-0.001621	-0.000755	0.044167	0.420464	-0.082397
	0.6	0.050952	0.035206	0.420844	0.416880	0.047469	0.413994	0.000000	0.050796	0.044569	0.438380	0.429167	-0.000347	-0.000699	0.088240	0.439109	0.000000
	0.7	0.425094	0.421348	0.416095	0.415244	0.455056	0.419097	0.000000	0.432584	0.432584	0.429055	0.437743	-0.000566	0.002730	0.453933	0.431231	0.000000
	0.8	-0.005618	0.005618	0.001484	0.007347	0.005618	0.001516	0.000000	0.000000	0.000000	0.017796	0.011091	-0.000699	0.001928	0.000000	0.006902	0.000000
	0.9	0.000000	0.000000	0.008105	0.006167	0.000000	0.008841	0.000000	0.000000	0.000000	0.012779	0.010367	0.001928	0.000268	0.000000	0.013051	0.000000
	1.0	0.000000	0.000000	0.008105	0.006311	0.000000	0.008105	0.000000	0.000000	0.000000	0.012255	0.010367	0.000000	0.000000	0.000000	0.012255	0.000000
recall	0.0	0.092697	0.089687	0.049340	0.044358	0.093098	0.048363	0.244817	0.108246	0.097913	0.046800	0.040059	-0.006351	-0.007245	0.105237	0.043478	0.430578
	0.1	0.092697	0.042536	0.049340	0.027064	0.093098	0.048363	0.060018	0.108246	0.077047	0.046800	0.029213	-0.006351	-0.007198	0.105237	0.043478	0.227729
	0.2	0.092697	0.041031	0.049340	0.025696	0.093098	0.048363	0.021849	0.108246	0.070325	0.046800	0.027748	-0.006351	-0.007904	0.105237	0.043478	-0.022171
	0.3	0.092496	0.033909	0.049340	0.023547	0.093098	0.048363	0.005371	0.107343	0.047552	0.046800	0.024231	-0.006351	-0.008898	0.105237	0.043478	-0.012239
	0.4	0.091091	0.031100	0.049145	0.020713	0.093098	0.048363	0.000858	0.101124	0.033106	0.046605	0.020029	-0.006448	-0.010505	0.105237	0.043478	-0.003812
	0.5	0.083266	0.029294	0.046312	0.019150	0.093098	0.048363	-0.000116	0.069322	0.029896	0.041817	0.018271	-0.007914	-0.012877	0.105237	0.043478	-0.002207
	0.6	0.000301	0.000502	0.016707	0.010259	0.000301	0.017782	0.000000	0.003913	0.004213	0.017587	0.011041	-0.008793	-0.013137	0.004514	0.019052	0.000000
	0.7	0.003913	0.003812	0.005960	0.001466	0.004314	0.006448	0.000000	0.003913	0.003913	0.005765	0.001368	-0.011138	-0.012752	0.004314	0.006351	0.000000
	0.8	0.000000	0.000100	0.008696	0.003908	0.000100	0.009086	0.000000	0.000000	0.000000	0.009086	0.003810	-0.013385	-0.010940	0.000000	0.009282	0.000000
	0.9	0.000000	0.000000	0.006058	0.003029	0.000000	0.006546	0.000000	0.000000	0.000000	0.006448	0.002931	-0.010943	-0.007109	0.000000	0.006546	0.000000
	1.0	0.000000	0.000000	0.006058	0.003029	0.000000	0.006058	0.000000	0.000000	0.000000	0.006058	0.002931	0.000000	0.000000	0.000000	0.006058	0.000000

Table A.9.: Results of MUDetect with MUBench-on-MUBench and MUBench-on-AU500 setting

Dataset	Support	#tp	#fp	#tn	#fn	Precision (%)	Recall (%)
MUBench	6	73	42	71	40	63.5	64.6
MUBench	11	58	27	86	55	68.2	51.3
MUBench	21	56	20	93	57	73.7	49.6
MUBench	41	0	1	112	113	0.0	0.0
AU500	5	38	158	227	77	19.4	33.0
AU500	10	18	32	353	97	36.0	15.7
AU500	20	17	28	357	98	37.8	14.8
AU500	40	10	17	368	105	37.0	8.7

A.4.1. Further Comparison RuDetect and MUDetect

We provide two further comparisons between Change Rule-based API Misuse Detection (RuDetect) and MUDetect by providing the detailed results with multiple *support*- (i.e., for MUDetect) and *threshold*-values (i.e., for RuDetect) based on both datasets (i.e., MUBench and AU500). Moreover, we present detailed results of RuDetect when *applying change rules without context*.

Detailed Comparison Subsequently, we present the results of the comparison between RuDetect and MUDetect with the absolute number of true positive (tp), false positive (fp), true negatives (tn), and false negatives (tn) for both datasets (i.e., MUBench and AU500). In detail, Table A.9 describes all results obtained by MUDetect, Tables A.10 and A.11 show the results of RuDetect on MUBench using *threshold* 0.6 and 0.7, respectively, and Tables A.12 and A.13 present the same but for the AU500 dataset.

Table A.10.: Results of selected change rules with $threshold = 0.6$ for misuse detection on the MUBench-on-MUBench experiment

Similarity	#Entries$_{M/C}$	#tp	#fp	#tn	#fn	Precision (%)	Recall (%)
APIExasVectorCosine	113	62	41	72	51	60.2	54.9
APIExasVectorL1Norm	113	60	39	74	53	60.6	53.1
APIExasVectorSplitCosine	113	23	20	93	90	53.5	20.4
APIExasVectorSplitL1Norm	113	20	15	98	93	57.1	17.7
APIIndicatorExasVector	113	62	4	109	51	93.9	54.9
APIIndicatorExasVectorSplit	113	23	21	92	90	52.3	20.4
AStarGED	113	0	0	113	113	-	0.0
ExasVectorCosine	113	60	41	72	53	59.4	53.1
ExasVectorL1Norm	113	59	40	73	54	59.6	52.2
ExasVectorSplitCosine	113	25	16	97	88	61.0	22.1
ExasVectorSplitL1Norm	113	23	16	97	90	59.0	20.4
HungarianGED	113	0	0	113	113	-	0.0
HungarianMCS	113	0	0	113	113	-	0.0
IndicatorExasVector	113	60	4	109	53	93.8	53.1
IndicatorExasVectorSplit	113	24	15	98	89	61.5	21.2
NodeSimilarityOpt	113	0	0	113	113	-	0.0

Table A.11.: Results of selected change rules with $threshold = 0.7$ for misuse detection on the MUBench-on-MUBench experiment

Similarity	#Entries	#tp	#fp	#tn	#fn	Precision (%)	Recall (%)
APIExasVectorCosine	113	56	38	75	57	59.6	49.6
APIExasVectorL1Norm	113	51	38	75	62	57.3	45.1
APIExasVectorSplitCosine	113	13	14	99	100	48.1	11.5
APIExasVectorSplitL1Norm	113	8	7	106	105	53.3	7.1
APIIndicatorExasVector	113	56	1	112	57	98.2	49.6
APIIndicatorExasVectorSplit	113	13	15	98	100	46.4	11.5
AStarGED	113	0	0	113	113	-	0.0
ExasVectorCosine	113	56	38	75	57	59.6	49.6
ExasVectorL1Norm	113	49	38	75	64	56.3	43.4
ExasVectorSplitCosine	113	19	12	101	94	61.3	16.8
ExasVectorSplitL1Norm	113	14	9	104	99	60.9	12.4
HungarianGED	113	0	0	113	113	-	0.0
HungarianMCS	113	0	0	113	113	-	0.0
IndicatorExasVector	113	56	1	112	57	98.2	49.6
IndicatorExasVectorSplit	113	18	11	102	95	62.1	15.9
NodeSimilarityOpt	113	0	0	113	113	-	0.0

Table A.12.: Results of selected change rules with $threshold = 0.6$ for misuse detection on the MUBench-on-AU500 experiment

Similarity	**#Entires$_M$**	**#Entries$_C$**	**#tp**	**#fp**	**#tn**	**#fn**	**Precision (%)**	**Recall (%)**
APIExasVectorCosine	115	385	4	9	376	111	30.8	3.5
APIExasVectorL1Norm	115	385	4	7	378	111	36.4	3.5
APIExasVectorSplitCosine	115	385	19	89	296	96	17.6	16.5
APIExasVectorSplitL1Norm	115	385	17	74	311	98	18.7	14.8
APIIndicatorExasVector	115	385	3	16	369	112	15.8	2.6
APIIndicatorExasVectorSplit	115	385	20	85	300	95	19.0	17.4
AStarGED	115	385	0	0	385	115	-	0.0
ExasVectorCosine	115	385	5	8	377	110	38.5	4.3
ExasVectorL1Norm	115	385	5	6	379	110	45.5	4.3
ExasVectorSplitCosine	115	385	21	97	288	94	17.8	18.3
ExasVectorSplitL1Norm	115	385	20	86	299	95	18.9	17.4
HungarianGED	115	385	0	0	385	115	-	0.0
HungarianMCS	115	385	0	0	385	115	-	0.0
IndicatorExasVector	115	385	5	22	363	110	18.5	4.3
IndicatorExasVectorSplit	115	385	23	93	292	92	19.8	20.0
NodeSimilarityOpt	115	385	0	0	385	115	-	0.0

Table A.13.: Results of selected change rules with $threshold = 0.7$ for misuse detection on the MUBench-on-AU500 experiment

Similarity	**#Entires$_M$**	**#Entries$_C$**	**#tp**	**#fp**	**#tn**	**#fn**	**Precision (%)**	**Recall (%)**
APIExasVectorCosine	115	385	3	2	383	112	60.0	2.6
APIExasVectorL1Norm	115	385	2	2	383	113	50.0	1.7
APIExasVectorSplitCosine	115	385	15	60	325	100	20.0	13.0
APIExasVectorSplitL1Norm	115	385	10	43	342	105	18.9	8.7
APIIndicatorExasVector	115	385	3	3	382	112	50.0	2.6
APIIndicatorExasVectorSplit	115	385	17	60	325	98	22.1	14.8
AStarGED	115	385	0	0	385	115	-	0.0
ExasVectorCosine	115	385	3	1	384	112	75.0	2.6
ExasVectorL1Norm	115	385	2	1	384	113	66.7	1.7
ExasVectorSplitCosine	115	385	19	67	318	96	22.1	16.5
ExasVectorSplitL1Norm	115	385	14	48	337	101	22.6	12.2
HungarianGED	115	385	0	0	385	115	-	0.0
HungarianMCS	115	385	0	0	385	115	-	0.0
IndicatorExasVector	115	385	3	4	381	112	42.9	2.6
IndicatorExasVectorSplit	115	385	21	67	318	94	23.9	18.3
NodeSimilarityOpt	115	385	0	0	385	115	-	0.0

Table A.14.: Results of selected change rules *without context* for misuse detection on the MUBench-on-MUBench experiment

Similarity	#Entries	#tp	#fp	#tn	#fn	Precision (%)	Recall (%)
APIExasVectorCosine	113	49	3	110	64	94.2	43.4
APIExasVectorL1Norm	113	11	3	110	102	78.6	9.7
APIExasVectorSplitCosine	113	58	16	97	55	78.4	51.3
APIExasVectorSplitL1Norm	113	20	16	97	93	55.6	17.7
APIIndicatorExasVector	113	49	3	110	64	94.2	43.4
APIIndicatorExasVectorSplit	113	59	16	97	54	78.7	52.2
AStarGED	113	0	0	113	113	-	0.0
ExasVectorCosine	113	48	1	112	65	98.0	42.5
ExasVectorL1Norm	113	10	2	111	103	83.3	8.8
ExasVectorSplitCosine	113	59	13	100	54	81.9	52.2
ExasVectorSplitL1Norm	113	20	13	100	93	60.6	17.7
HungarianGED	113	0	0	113	113	-	0.0
HungarianMCS	113	0	0	113	113	-	0.0
IndicatorExasVector	113	49	2	111	64	96.1	43.4
IndicatorExasVectorSplit	113	61	13	100	52	82.4	54.0
NodeSimilarityOpt	113	2	0	113	111	100.0	1.8

Comparison Using Rules Without Context We considered the influence of using the *context* (i.e., comparing rules inferred *with* and *without single-hop-addition*). We present the results for the change rule-based misuse detection for both setting, namely MUBench-on-MUBench and MUBench-on-AU500, in Table A.14 and Table A.15. In the MUBench-on-MUBench settings (cf. Table A.14) we found that all but two Exas-vector similarity (i.e., exceptions are the `SplitL1Norm`-variants) obtain better results in terms of precision than MUDetect ranging from $78.4 - 98\%$ with varying recall between $9.8 - 54.5\%$. Additionally, `NodeSimilarityOpt` resulted in a perfect precision, however, it is accompanied by a very low recall of 1.8%. All other non-Exas-vector variants have no positive results. While this result may indicate that using *context* in change rules hinders precise misuse detection, we observe from the MUBench-on-AU500 setting (cf. Table A.15) an opposite result. In this evaluation, we observe that all similarities perform worse than MUDetect in terms of precision. Nevertheless, all but one `-Split-`variant (i.e., exception by `ExasVectorSplitL1Norm`) obtains similar or better recall values than MUDetect with *support* value 20, but worse compared to MUDetect at *support* value 5 with, however, a partially lower precision. Therefore, we conclude the results as follows:

1. Usually, context enables change-rule-based misuse detection to obtain better results in terms of precision in a cross-project setting.

2. Change-rule-based misuse detection with `-Split-` can benefit from not using the context to obtain better precision and recall in the cross-project setting.

Table A.15.: Results of selected change rules *without context* for misuse detection on the MUBench-on-AU500 experiment

Similarity	**#Entires$_M$**	**#Entries$_C$**	**#tp**	**#fp**	**#tn**	**#fn**	**Precision (%)**	**Recall (%)**
APIExasVectorCosine	115	385	5	37	348	110	11.9	4.3
APIExasVectorL1Norm	115	385	2	5	380	113	28.6	1.7
APIExasVectorSplitCosine	115	385	27	95	290	88	22.1	23.5
APIExasVectorSplitL1Norm	115	385	24	80	305	91	23.1	20.9
APIIndicatorExasVector	115	385	5	47	338	110	9.6	4.3
APIIndicatorExasVectorSplit	115	385	27	104	281	88	20.6	23.5
AStarGED	115	385	0	0	385	115	-	0.0
ExasVectorCosine	115	385	5	35	350	110	12.5	4.3
ExasVectorL1Norm	115	385	2	6	379	113	25.0	1.7
ExasVectorSplitCosine	115	385	20	88	297	95	18.5	17.4
ExasVectorSplitL1Norm	115	385	16	68	317	99	19.0	13.9
HungarianGED	115	385	0	0	385	115	-	0.0
HungarianMCS	115	385	0	0	385	115	-	0.0
IndicatorExasVector	115	385	5	47	338	110	9.6	4.3
IndicatorExasVectorSplit	115	385	22	93	292	93	19.1	19.1
NodeSimilarityOpt	115	385	0	0	385	115	-	0.0

Bibliography

[AARRM15] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems. In *Proceedings of the 4th International Conference on Pattern Recognition Applications and Methods (ICPRAM)*, pages 271–278. SciTePress, 2015. doi:10.5220/0005209202710278. (cited on pages 154, 217, and 220.)

[AAWX23] Hadeel Alhosaini, Sultan Alharbi, Xianzhi Wang, and Guandong Xu. API Recommendation For Mashup Creation: A Comprehensive Survey. *The Computer Journal*, 67(5):1920–1940, 11 2023. doi:10.1093/comjnl/bxad112. (cited on pages 70 and 71.)

[Abd07] Hervé Abdi. *Encyclopedia of Measurement and Statistics*, volume 1, chapter Bonferroni Test, pages 103–107. SAGE, 2007. doi:10.4135/9781412952644.n60. (cited on pages 120, 163, 173, 186, and 194.)

[ABF+16] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *Proceedings of the 37th IEEE Security & Privacy*, pages 289–305. IEEE, 2016. doi:10.1109/SP.2016.25. (cited on pages 54 and 253.)

[ABF+17] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the Usability of Cryptographic APIs. In *Proceedings of the 38th IEEE Security & Privacy*, pages 154–171. IEEE, 2017. doi:10.1109/SP.2017.52. (cited on pages 4, 54, 78, 251, and 258.)

[ABH14] Charu C. Aggarwal, Mansurul A. Bhuiyan, and Mohammad Al Hasan. *Frequent Pattern Mining Algorithms: A Survey*, pages 19–64. Springer, Cham, 2014. doi:10.1007/978-3-319-07821-2_2. (cited on pages 36, 37, and 38.)

[ABKLT16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: collecting millions of Android apps for the research community. In *Proceedings of the 13th International Working Conference on Mining Software Repositories (MSR)*, pages 468–471. ACM, 2016. doi:10.1145/2901739.2903508. (cited on pages 97 and 105.)

[Afo15] Luiz Marques Afonso. *Communicative Dimensions of Application Programming Interfaces (APIs)*. PhD thesis, Pontifícia Universidade Católica Do Rio De Janeiro, Brazil, 2015. URL: http://www-di.inf.puc-rio.br/~clarisse/docs/2015TeseDoutoradoLuizMarques.pdf. (cited on pages 54, 59, 250, and 254.)

[Agg14] Charu C. Aggarwal. *An Introduction to Frequent Pattern Mining*, pages 1–17. Springer, Cham, 2014. doi:10.1007/978-3-319-07821-2_1. (cited on pages 36 and 37.)

[Agg15] Charu C. Aggarwal. *Data Preparation*, pages 27–62. Springer, Cham, 2015. doi:10.1007/978-3-319-14142-8_2. (cited on pages 37 and 41.)

[AH14] Charu C. Aggarwal and Jiawei Han, editors. *Frequent Pattern Mining*. Springer, 1 edition, 2014. doi:10.1007/978-3-319-07821-2. (cited on page 36.)

[All70] Frances E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19. ACM, 1970. doi:10.1145/800028.808479. (cited on page 28.)

[ALSU14] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education Limited, 2 edition, 2014. ISBN-13: 978-1-292-02434-9. (cited on pages 27 and 28.)

[AM18] Amritanshu Agrawal and Tim Menzies. Is "Better Data" Better Than "Better Data Miners"?: On the Benefits of Tuning SMOTE for Defect Prediction. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 1050–1061. ACM, 2018. doi:10.1145/3180155.3180197. (cited on page 109.)

[Ama18] Sven Amann. *A Systematic Approach to Benchmark and Improve Automated Static Detection of Java-API Misuses*. PhD thesis, Darmstadt University of Technology, Germany, 2018. URL: http://tuprints.ulb.tu-darmstadt.de/7422/. (cited on pages 28, 40, 42, 43, 103, 106, 107, 109, 114, 132, 136, 139, 141, and 242.)

[ANBL18] E. Aghajani, C. Nagy, G. Bavota, and M. Lanza. A Large-Scale Empirical Study on Linguistic Antipatterns Affecting APIs. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME)*, pages 25–35. IEEE, 2018. doi:10.1109/ICSME.2018.00012. (cited on pages 4, 54, 59, 78, 253, and 258.)

[ANN+16] Sven Amann, Sarah Nadi, Hoan Anh Nguyen, Tien N. Nguyen, and Mira Mezini. MUBench: A Benchmark for API-Misuse Detectors. In *Proceedings of the 13th International Working Conference on Mining Software Repositories (MSR)*, pages 464–467. ACM, 2016. doi:10.1145/2901739.2903506. (cited on pages 2, 26, 27, 97, 112, 114, 115, 117, 132, 137, 149, 157, 163, 179, 196, 197, 202, 209, 220, 221, 223, 236, 240, and 244.)

[ANN+19a] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering (TSE)*, 45(12):1170–1188, 2019. doi:10.1109/TSE.2018.2827384. (cited on pages 2, 3, 5, 14, 26, 27, 28, 36, 40, 93, 108, 109, 112, 117, 140, 143, and 144.)

[ANN+19b] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. Investigating next Steps in Static API-Misuse Detection. In *Proceedings of the 16th International Working Conference on Mining Software Repositories (MSR)*, pages 265–275. IEEE, 2019. doi:10.1109/MSR.2019.00053. (cited on pages 5, 14, 17, 28, 42, 43, 93, 95, 97,

98, 99, 101, 106, 107, 132, 136, 137, 138, 140, 141, 144, 145, 161, 184, 185, 186, 197, 198, 199, 200, 202, 203, 222, 223, 233, 234, 236, 239, 242, and 267.)

[APH+08] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008. doi:10.1109/MS.2008.130. (cited on page 25.)

[ARB20] George Ajam, Carlos Rodríguez, and Boualem Benatallah. API Topics Issues in Stack Overflow Q&As Posts: An Empirical Study. In *Proceedings of the 46th Latin American Computing Conference (CLEI)*, pages 147–155. IEEE, 2020. doi:10.1109/CLEI52000.2020.00024. (cited on pages 54 and 251.)

[Arv18] Svante Arvedahl. Introducing debtgrep, a tool for fighting technical debt in base station software. In *Proceedings of the 1st International Conference on Technical Debt (TechDebt)*, pages 51–52. ACM, 2018. doi:10.1145/3194164.3194183. (cited on page 35.)

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499. Morgan Kaufmann Publishers Inc., 1994. (cited on page 38.)

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining Sequential Patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, pages 3–14. IEEE, 1995. doi:10.1109/ICDE.1995.380415. (cited on page 39.)

[AS14] Miltiadis Allamanis and Charles Sutton. Mining Idioms from Source Code. In *Proceedings of the 22nd International Symposium Foundations of Software Engineering (FSE)*, pages 472–483. ACM, 2014. doi:10.1145/2635868.2635901. (cited on page 109.)

[ATLJ20] Muhammad Hilmi Asyrofi, Ferdian Thung, David Lo, and Lingxiao Jiang. AUSearch: Accurate API Usage Search in GitHub Repositories with Type Resolution. In *Proceedings of the 27th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 637–641. IEEE, 2020. doi:10.1109/SANER48275.2020.9054809. (cited on pages 32, 104, and 138.)

[AX09] Mithun Acharya and Tao Xie. Mining API Error-Handling Specifications from Source Code. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 5503, pages 370–384. Springer, 2009. doi:10.1007/978-3-642-00593-0_25. (cited on pages 36, 96, 99, 100, 101, and 267.)

[Bay98] Roberto J. Bayardo. Efficiently Mining Long Patterns from Databases. In *Proceedings of the 25th ACM SIGMOD International Conference on Management of Data (MOD)*, pages 85–93. ACM, 1998. doi:10.1145/276304.276313. (cited on page 38.)

[BB13] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013. doi:10.1109/ICSE.2013.6606617. (cited on page 4.)

[BBR03] Jean-François Boulicaut, Artur Bykowski, and Christophe Rigotti. Free-Sets: A Condensed Representation of Boolean Data for the Approximation of Frequency Queries. *Springer Data Mining and Knowledge Discovery*, 7(1):5–22, 2003. doi:10.1023/A:1021571501451. (cited on page 40.)

[BBZJ14] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix? In *Proceedings of the 11th International Working Conference on Mining Software Repositories (MSR)*, pages 202–211. ACM, 2014. doi:10.1145/2597073.2597082. (cited on page 25.)

[BCH+23] Avinash Bhat, Austin Coursey, Grace Hu, Sixian Li, Nadia Nahar, Shurui Zhou, Christian Kästner, and Jin L.C. Guo. Aspirations and Practice of ML Model Documentation: Moving the Needle with Nudging and Traceability. In *Proceedings of the 41st Conference on Human Factors in Computing Systems (CHI)*, pages 1–17. ACM, 2023. doi:10.1145/3544548.3581518. (cited on pages 79 and 261.)

[BCHH10] Renée C. Bryce, Alison Cooley, Amy Hansen, and Nare Hayrapetyan. A One Year Empirical Study of Student Programming Bugs. In *2010 IEEE Frontiers in Education Conference (FIE)*, pages F1G–1–F1G–7. IEEE, 2010. doi:10.1109/FIE.2010.5673143. (cited on page 24.)

[BCM22] Gloria Bondel, Arif Cerit, and Florian Matthes. Challenges of API Documentation from a Provider Perspective and Best Practices for Examples in Public Web API Documentation. In *Proceedings of the 24th International Conference on Enterprise Information Systems (ICEIS)*, pages 268–279. SciTePress, 2022. doi:10.5220/0011089700003179. (cited on pages 54, 79, 252, 255, and 260.)

[BDWK10] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-Centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the 28th Conference on Human Factors in Computing Systems (CHI)*, pages 513–522. ACM, 2010. doi:10.1145/1753326.1753402. (cited on pages 78, 79, 259, and 260.)

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002. ISBN-13: 978-0321146533. (cited on page 4.)

[BEH+23] Maria Teresa Baldassarre, Neil Ernst, Ben Hermann, Tim Menzies, and Rahul Yedida. (re)use of research results (is rampant). *Communications of the ACM*, 66(2):75–81, January 2023. doi:10.1145/3554976. (cited on page 245.)

[BFH24] Nick C. Bradley, Thomas Fritz, and Reid Holmes. Supporting Web-Based API Searches in the IDE Using Signatures. In *Proceedings of the 46th*

International Conference on Software Engineering (ICSE), pages 1–12. ACM, 2024. doi:10.1145/3597503.3639089. (cited on pages 78, 79, 259, and 260.)

[BFHM12] Chris Burns, Jennifer Ferreira, Theodore D. Hellmann, and Frank Maurer. Usable results from the field of API usability: A systematic mapping and further analysis. In *Proceedings of the 29th IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 179–182. IEEE, 2012. doi:10.1109/VLHCC.2012.6344511. (cited on pages 4, 46, 47, 70, and 71.)

[BFSK20] Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), November 2020. doi:10.1145/3428282. (cited on pages 24 and 54.)

[BGH+04] Vincent D. Blondel, Anahí Gajardo, Maureen Heymans, Pierre Senellart, and Paul Van Dooren. A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching. *SIAM Review*, 46(4):647–666, April 2004. doi:10.1137/S0036144502415960. (cited on page 154.)

[BGH+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 169–190. ACM, 2006. doi:10.1145/1167473.1167488. (cited on page 97.)

[BHS15] Aurélien Bellet, Amaury Habrard, and Marc Sebban. *Metric Learning*. Morgan & Claypool Publishers, February 2015. ISBN-13: 978-3-031-00444-5. doi:10.1007/978-3-031-01572-4. (cited on page 248.)

[BHVR16] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. Do Developers Deprecate APIs with Replacement Messages? A Large-Scale Analysis on Java Systems. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 360–369. IEEE, 2016. doi:10.1109/SANER.2016.99. (cited on page 34.)

[BMG12] Mario Boley, Sandy Moens, and Thomas Gärtner. Linear Space Direct Pattern Sampling using Coupling From The Past. In *Proceedings of the 18th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 69–77. ACM, 2012. doi:10.1145/2339530.2339545. (cited on page 40.)

[BOST22] Wilson Baker, Michael O'Connor, Seyed Reza Shahamiri, and Valerio Terragni. Detect, Fix, and Verify TensorFlow API Misuses. In *Proceedings of the 29th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 925–929. IEEE, 2022. doi:10.1109/SANER53432.2022.00110. (cited on pages 209, 213, 230, and 231.)

[BR22] Leif Bonorden and Matthias Riebisch. API Deprecation: A Systematic Mapping Study. In *Proceedings of the 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 451–458. IEEE, 2022. doi:10.1109/SEAA56994.2022.00076. (cited on pages 34, 46, 47, 70, and 102.)

[BS98] Horst Bunke and Kim Shearer. A graph distance metric based on the maximal common subgraph. *Elsevier Pattern Recognition Letters (PRLEDG)*, 19(3):255–259, 1998. doi:10.1016/S0167-8655(97)00179-7. (cited on page 154.)

[BSPC19] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to Fix Bugs Automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), October 2019. doi:10.1145/3360585. (cited on pages 18, 207, 215, 231, 234, and 247.)

[BSvdB15] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Eclipse API usage: the good and the bad. *Software Quality Journal (SQJ)*, 23(1):107–141, 2015. doi:10.1007/S11219-013-9221-3. (cited on pages 54 and 251.)

[BSZ+20] Celeste Barnaby, Koushik Sen, Tianyi Zhang, Elena Glassman, and Satish Chandra. Exempla Gratis (E.G.): Code Examples for Free. In *Proceedings of the 28th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 1353–1364. ACM, 2020. doi:10.1145/3368089.3417052. (cited on pages 78, 79, 259, and 260.)

[BVXH20] Aline Brito, Marco Túlio Valente, Laerte Xavier, and André C. Hora. You broke my code: understanding the motivations for breaking changes in APIs. *Springer Empirical Software Engineering (EMSE)*, 25(2):1458–1492, November 2020. doi:10.1007/S10664-019-09756-Z. (cited on pages 34 and 54.)

[BW12] Raymond P. L. Buse and Westley Weimer. Synthesizing API Usage Examples. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 782–792. IEEE, 2012. doi:10.1109/ICSE.2012.6227140. (cited on pages 79 and 260.)

[BXHV18] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. Why and how Java developers break APIs. In *Proceedings of the 25th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 255–265. IEEE, 2018. doi:10.1109/SANER.2018.8330214. (cited on pages 54 and 254.)

[BZ06] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 158–168. ACM, 2006. doi:10.1145/1133981.1134000. (cited on page 210.)

[Car98] D.N. Card. Learning from our mistakes with defect causal analysis. *IEEE Software*, 15(1):56–63, 1998. doi:10.1109/52.646883. (cited on page 4.)

[CCL12] Wing-Kwan Chan, Hong Cheng, and David Lo. Searching Connected API Subgraph via Text phrases. In *Proceedings of the 20th International Symposium Foundations of Software Engineering (FSE)*. ACM, 2012. doi:10.1145/2393596.2393606. (cited on page 32.)

[CDAR22] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. CODIT: Code Editing With Tree-Based Neural Models. *IEEE Transactions on Software Engineering (TSE)*, 48(4):1385–1399, 2022. doi:10.1109/TSE.2020.3020502. (cited on page 212.)

[CFW24] Adele E. Clarke, Carrie Friese, and Rachel Washburn. *The SAGE Handbook of Qualitative Research*, chapter Critical Situational Analysis After the Interpretive Turn. SAGE, Los Angeles, 6 edition, 2024. edited by Norman K. Denzin (University of Illinois Urbana-Champaign), Yvonna S. Lincoln (Texas A&M University), Michael D. Giardina (Florida State University), Gaile S. Cannella (Independent Scholar). (cited on pages 48 and 51.)

[CGM+13] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic Recovery from Runtime Failures. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 782–791. IEEE, 2013. doi:10.1109/ICSE.2013.6606624. (cited on page 210.)

[CJCaK+00] Pete Chapman, Randy Kerber Julian Clinton and, Thomas Khabaza, Thomas Reinartz, Colin Shearer, and Rüdiger Wirth. CRISP-DM 1.0 Step-by-step data mining guide, August 2000. CRISP-DM consortium: NCR Systems Engineering Copenhagen (USA and Denmark), Daimler-Chrysler AG (Germany), SPSS Inc. (USA), and OHRA Verzekeringen en Bank Groep B.V (The Netherlands). (cited on pages 41 and 42.)

[CKF+04] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot—A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, pages 31–44. USENIX, December 2004. URL: http://usenix.org/publications/library/proceedings/osdi04/tech/full_papers/candea/candea.pdf. (cited on page 210.)

[CM04] B. Chess and G. McGraw. Static Analysis for Security. *IEEE Security & Privacy*, 2(6):76–79, 2004. doi:10.1109/MSP.2004.111. (cited on pages 5 and 108.)

[ÇM18] Ervina Çergani and Mira Mezini. On the Impact of Order Information in API Usage Patterns. In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT)*, volume 1077, pages 79–103. Springer, 2018. doi:10.1007/978-3-030-29157-0_4. (cited on pages 5, 93, 96, 97, 99, 101, 102, 109, and 267.)

[CNSH14] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. An Empirical Study of Dormant Bugs. In *Proceedings of the 11th International Working Conference on Mining Software Repositories (MSR)*, pages 82–91. ACM, 2014. doi:10.1145/2597073.2597108. (cited on page 24.)

[Coh60] Jacob Cohen. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. doi:10.1177/001316446002000104. (cited on page 149.)

[COZ21] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. Fast and Precise On-the-Fly Patch Validation for All. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 1123–1134. IEEE, 2021. doi:10.1109/ICSE43902.2021.00104. (cited on page 208.)

[CPS24] Satish Chandra, Michael Pradel, and Kathryn T. Stolee. Code Search (Dagstuhl Seminar 24172). *Dagstuhl Reports*, 14(4):108–123, 2024. doi:10.4230/DagRep.14.4.108. (cited on page 32.)

[CPY+24] Xiang Chen, Wenlong Pei, Shaoyu Yang, Yanlin Zhou, Zichen Zhang, and Jiahua Pei. Automatic title completion for Stack Overflow posts and GitHub issues. *Springer Empirical Software Engineering (EMSE)*, 29(5):120, 2024. doi:10.1007/S10664-024-10513-0. (cited on pages 78, 258, and 259.)

[Cro16] David F. Crouse. On implementing 2d rectangular assignment algorithms. *IEEE Transactions on Aerospace and Electronic Systems*, 52(4):1679–1696, 2016. doi:10.1109/TAES.2016.140952. (cited on page 154.)

[CRPA12] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: An Eclipse Plug-in for Testing and Debugging. In *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*, pages 378–381. ACM, 2012. doi:10.1145/2351676.2351752. (cited on page 208.)

[CS14] Marcelo Cataldo and Cleidson R. B. De Souza. Exploring the Impact of API Complexity on Failure-Proneness. In *Proceedings of the 9th International Conference on Global Software Engineering (ICGSE)*, pages 36–45. IEEE, 2014. doi:10.1109/ICGSE.2014.16. (cited on pages 54, 250, and 252.)

[CS25] Scott Chacon and Ben Straub. *Pro Git*. Number Version 2.1.447. Apress, April 2025. 2025-04-10. URL: https://github.com/progit/progit2/releases/tag/2.1.447. (cited on page 33.)

[CSS13] K.K. Chaturvedi, V.B. Sing, and Prashast Singh. Tools in Mining Software Repositories. In *Proceedings of the 13th International Conference on Computational Science and Its Applications (ICCSA)*, pages 89–98. IEEE, 2013. doi:10.1109/ICCSA.2013.22. (cited on page 34.)

[CVG19] Alex Cummaudo, Rajesh Vasa, and John Grundy. What should I document? A preliminary systematic mapping study into API documentation knowledge. In *Proceedings of the 13th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6. IEEE, 2019. doi:10.1109/ESEM.2019.8870148. (cited on pages 46, 47, 65, 70, and 71.)

[CXL19] Chunyang Chen, Zhenchang Xing, and Yang Liu. What's Spain's Paris? Mining analogical libraries from Q&A discussions. *Springer Empirical Software Engineering (EMSE)*, 24(3):1155–1194, 2019. doi:10.1007/S10664-018-9657-Y. (cited on pages 78, 79, 260, and 261.)

[CXLX21a] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. Mining Likely Analogical APIs Across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Transactions on Software Engineering (TSE)*, 47(3):432–447, 2021. doi:10.1109/TSE.2019.2896123. (cited on page 32.)

[CXLX21b] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. Mining Likely Analogical APIs Across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Transactions on Software Engineering (TSE)*, 47(3):432–447, 2021. doi:10.1109/TSE.2019.2896123. (cited on pages 35 and 199.)

[CYH14] Hong Cheng, Xifeng Yan, and Jiawei Han. *Mining Graph Patterns*, pages 307–338. Springer, Cham, 2014. doi:10.1007/978-3-319-07821-2_13. (cited on pages 36, 39, and 40.)

[CYYZ19] Cong Chen, Yulong Yang, Lin Yang, and Kang Zhang. A Human-as-Sensors Approach to API Documentation Integration and Its Effects on Novice Programmers. In *Proceedings of the 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 197–206. IEEE, 2019. doi:10.1109/SANER.2019.8668026. (cited on pages 79 and 260.)

[CZLF19] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. A large-scale study of application incompatibilities in Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 216–227. ACM, 2019. doi:10.1145/3293882.3330564. (cited on pages 54, 252, 253, and 254.)

[DCSM17] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Dynamic Patch Generation for Null Pointer Exceptions using Metaprogramming. In *Proceedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 349–358. IEEE, 2017. doi:10.1109/SANER.2017.7884635. (cited on page 211.)

[DER12] Ekwa Duala-Ekoko and Martin P. Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 266–276. IEEE, 2012. doi:10.1109/ICSE.2012.6227187. (cited on pages 54, 250, 251, and 254.)

[DGP23] Luca Di Grazia and Michael Pradel. Code Search: A Survey of Techniques for Finding Code. *ACM Computing Surveys*, 55(11), February 2023. doi:10.1145/3565971. (cited on page 32.)

[DH09a] Uri Dekel and James D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 320–330. IEEE, 2009. doi:10.1109/ICSE.2009.5070532. (cited on pages 78, 79, 259, and 261.)

[DH09b] Uri Dekel and James D. Herbsleb. Reading the documentation of invoked API functions in program comprehension. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC)*, pages 168–177. IEEE, 2009. doi:10.1109/ICPC.2009.5090040. (cited on pages 54, 79, 253, and 260.)

[DJ05] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 389–398. IEEE, 2005. doi:10.1109/ICSM.2005.90. (cited on page 35.)

[DLMK10] Tuan-Anh Doan, David Lo, Shahar Maoz, and Siau-Cheng Khoo. LM: A Miner for Scenario-Based Specifications. In *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE)*, pages 319–320. ACM, 2010. doi:10.1145/1810295.1810370. (cited on page 5.)

[DLZ+22] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. FIRA: Fine-Grained graph-Based Code Change Representation for Automated Commit Message Generation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 970–981. ACM, 2022. doi:10.1145/3510003.3510069. (cited on page 199.)

[DM16] Thomas Durieux and Martin Monperrus. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST)*, pages 85–91. ACM, 2016. doi:10.1145/2896921.2896931. (cited on page 211.)

[DND+25] Juri Di Rocco, Phuong T. Nguyen, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. DeepMig: A transformer-based approach to support coupled library and code migrations. *Elsevier Journal of Information and Software Technology (IST)*, 177(107588), 2025. doi:10.1016/j.infsof.2024.107588. (cited on pages 35, 79, 199, and 261.)

[DNMJ08] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. ReBA: Refactoring-aware Binary Adaptation of Evolving Libraries. In *Proceedings of the 30th International Conference on Software Engineering*

(ICSE), pages 441–450. ACM, 2008. doi:10.1145/1368088.1368148. (cited on pages 79 and 261.)

[DNRN13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 422–431. IEEE, 2013. doi:10.1109/ICSE.2013.6606588. (cited on pages 34, 104, 116, and 138.)

[DNRS21] Anastasia Danilova, Alena Naiakshina, Anna Rasgauski, and Matthew Smith. Code Reviewing as Methodology for Online Security Studies with Developers - A Case Study with Freelancers on Password Storage. In *Proceedings of the 17th Symposium on Usable Privacy and Security (SOUPS)*, pages 397–416. USENIX, August 2021. URL: https://www.usenix.org/conference/soups2021/presentation/danilova. (cited on pages 78, 258, and 259.)

[DPCdAM16] Fernanda Madeiral Delfim, Klérisson Vinícius Ribeiro Paixão, Damien Cassou, and Marcelo de Almeida Maia. Redocumenting APIs with crowd knowledge: a coverage analysis based on question types. *Journal of the Brazilian Computer Society*, 22(1):9:1–9:34, 2016. doi:10.1186/S13173-016-0049-0. (cited on pages 79 and 260.)

[DPZ+22] Stéphane Ducasse, Guillermo Polito, Oleksandr Zaitsev, Marcus Denker, and Pablo Tesone. Deprewriter: On the fly rewriting method deprecations. *Journal of Object Technology*, 21(1):1:1–23, 2022. doi:10.5381/jot.2022.21.1.a1. (cited on page 35.)

[DR11] Ekwa Duala-Ekoko and Martin P. Robillard. Using Structure-Based Recommendations to Facilitate Discoverability in APIs. In Mira Mezini, editor, *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP)*, volume 6813 of *Lecture Notes in Computer Science*, pages 79–104. Springer, 2011. doi:10.1007/978-3-642-22655-7_5. (cited on pages 78 and 259.)

[DR14] Barthélémy Dagenais and Martin P. Robillard. Using Traceability Links to Recommend Adaptive Changes for Documentation Evolution. *IEEE Transactions on Software Engineering (TSE)*, 40(11):1126–1146, 2014. doi:10.1109/TSE.2014.2347969. (cited on page 35.)

[DW10] Vidroha Debroy and W. Eric Wong. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, pages 65–74. IEEE, 2010. doi:10.1109/ICST.2010.66. (cited on page 211.)

[DZM09] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating Fixes from Object Behavior Anomalies. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE)*, pages 550–554. IEEE, 2009. doi:10.1109/ASE.2009.15. (cited on pages 213, 230, and 231.)

[EBFK13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 73–84. ACM, 2013. doi:10.1145/2508859.2516693. (cited on page 100.)

[ECC01] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In Keith Marzullo and Mahadev Satyanarayanan, editors, *Proceedings of the18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72. ACM, 2001. doi:10.1145/502034.502041. (cited on page 5.)

[EH23] Christof Ebert and Lorin Hochstein. DevOps in Practice. *IEEE Software*, 40(1):29–36, 2023. doi:10.1109/MS.2022.3213285. (cited on page 4.)

[EHJ+21] Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. RAPID: Checking API usage for the Cloud in the Cloud. In *Proceedings of the 29th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 1416–1426. ACM, 2021. doi:10.1145/3468264.3473934. (cited on page 5.)

[EHRS14] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How do API Documentation and Static Typing Affect API Usability? In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 632–642. ACM, 2014. doi:10.1145/2568225.2568299. (cited on pages 80 and 261.)

[EM21] Zachary Eberhart and Collin McMillan. Dialogue Management for Interactive API Search. In *Proceedings of the 37th International Conference on Software Maintenance and Evolution (ICSME)*, pages 274–285. IEEE, 2021. doi:10.1109/ICSME52107.2021.00031. (cited on pages 78 and 259.)

[Epp99] David Eppstein. Subgraph Isomorphism in Planar Graphs and Related Problems. *Journal of Graph Algorithms and Applications*, 3(3):1–27, Jan. 1999. doi:10.7155/jgaa.00014. (cited on pages 40, 43, 125, and 217.)

[ESM07] Brian Ellis, Jeffrey Stylos, and Brad Myers. The Factory Pattern in API Design: A Usability Evaluation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 302–312. IEEE, 2007. doi:10.1109/ICSE.2007.85. (cited on pages 78, 258, and 259.)

[EZG14] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web API growing pains: Stories from client developers and their code. In *Proceedings of the 1st Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 84–93. IEEE, 2014. doi:10.1109/CSMR-WCRE.2014.6747228. (cited on page 34.)

[EZG15] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web API growing pains: Loosely coupled yet strongly tied. *Elsevier Journal of Systems and Software (JSS)*, 100:27–43, 2015. doi:10.1016/j.jss.2014.10.014. (cited on pages 54 and 253.)

[FA11] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 416–419. ACM, 2011. doi:10.1145/2025113.2025179. (cited on pages 26 and 208.)

[FBB+14] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*, volume 14. Addison Wesley Longman, Inc., November 2014. ISBN-13: 978-0-201-48567-7. (cited on pages 4 and 210.)

[FGH+23] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large Language Models for Software Engineering: Survey and Open Problems. In *Proceedings of the 45th International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023. doi:10.1109/ICSE-FoSE59343.2023.00008. (cited on page 247.)

[FGO17] Marco Filax, Tim Gonschorek, and Frank Ortmeier. Building Models We Can Rely On: Requirements Traceability for Model-Based Verification Techniques. In *Proceedings of the 5th International Symposium on Model-Based Safety and Assessment (IMBSA)*, pages 3–18. Springer, 2017. doi:10.1007/978-3-319-64119-5_1. (cited on page 4.)

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1536–1547. ACL, November 2020. doi:10.18653/v1/2020.findings-emnlp.139. (cited on page 212.)

[FHP+13] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 49–60. ACM, 2013. doi:10.1145/2508859.2516655. (cited on page 54.)

[Fli14] Uwe Flick. *An Introduction to Qualitative Research*. Sage, Los Angeles, Calif., 5 edition, 2014. ISBN-13: 978-1-4462-6778-3. (cited on pages 7, 12, 45, 48, 51, and 88.)

[FM24] Jean-Remy Falleri and Matias Martinez. Fine-grained, Accurate and Scalable Source Code Differencing. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, pages 1–12. ACM, 2024. doi:10.1145/3597503.3639148. (cited on page 199.)

[FMB+14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th International Conference on Auto-*

mated Software Engineering (ASE), pages 313–324. ACM, 2014. doi:10.1145/2642937.2642982. (cited on page 199.)

[FMGK24] Ehsan Firouzi, Ammar Mansuri, Mohammad Ghafari, and Maziar Kaveh. From Struggle to Simplicity with a Usable and Secure API for Encryption in Java. In *Proceedings of the 18th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 556–565. ACM, 2024. doi:10.1145/3674805.3695405. (cited on pages 77 and 258.)

[FWY+19] Qiang Fan, Tao Wang, Cheng Yang, Gang Yin, Yue Yu, and Huai min Wang. Why do they ask? An exploratory study of crowd discussions about Android application programming interface in stack overflow. *Journal of Central South University*, 26:2432–2446, October 2019. doi:10.1007/s11771-019-4185-5. (cited on page 54.)

[FWZ10] Umer Farooq, Leon Welicki, and Dieter Zirkler. API Usability Peer Reviews: A Method for Evaluating the Usability of Application Programming Interfaces. In *Proceedings of the 28th Conference on Human Factors in Computing Systems (CHI)*, pages 2327–2336. ACM, 2010. doi:10.1145/1753326.1753677. (cited on pages 78 and 258.)

[FXK+19] Felix Fischer, Huang Xiao, Ching-Yu Kao, Yannick Stachelscheid, Benjamin Johnson, Danial Razar, Paul Fawkesley, Nat Buckley, Konstantin Böttinger, Paul Muntean, and Jens Grossklags. Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography. In *Proceedings of the 28th USENIX Security Symposium*, pages 339–356. USENIX, August 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/fischer. (cited on pages 78, 258, and 259.)

[GALIF20] Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs. In *Proceedings of the 38th Conference on Human Factors in Computing Systems (CHI)*, pages 1–13. ACM, 2020. doi:10.1145/3313831.3376142. (cited on pages 4, 54, 251, and 258.)

[GAQ+18] Ioannis Gasparis, Azeem Aqil, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Rajiv Gupta, and Edward Colbert. Droid M+: Developer Support for Imbibing Android's New Permission Model. In *Proceedings of the 13th Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 765–776. ACM, 2018. doi:10.1145/3196494.3196533. (cited on pages 79 and 261.)

[GDCS22] Fatih Gurcan, Gonca Gokce Menekse Dalveren, Nergiz Ercil Cagiltay, and Ahmet Soylu. Detecting Latent Topics and Trends in Software Engineering Research Since 1980 Using Probabilistic Topic Modeling. *IEEE Access*, 10:74638–74654, 2022. doi:10.1109/ACCESS.2022.3190632. (cited on page 80.)

[GFX+10] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A Search Engine For Finding Highly Relevant Applications. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 475–484. ACM, 2010. doi:10.1145/1806799.1806868. (cited on pages 79 and 260.)

[GGM04] Floris Geerts, Bart Goethals, and Taneli Mielikäinen. Tiling Databases. In Einoshin Suzuki and Setsuo Arikawa, editors, *Proceedings of the 7th International Conference on Discovery Science (DS)*, volume 3245, pages 278–289. Springer, 2004. doi:10.1007/978-3-540-30214-8_22. (cited on page 40.)

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN-13: 978-0-201-63361-0. (cited on page 4.)

[GIW+18] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In *Proceedings of the 14th Symposium on Usable Privacy and Security (SOUPS)*, pages 265–281. USENIX, August 2018. URL: https://www.usenix.org/conference/soups2018/presentation/gorski. (cited on pages 54, 78, 258, and 259.)

[GK22] Nicolas E. Gold and Jens Krinke. Ethics in the mining of software repositories. *Springer Empirical Software Engineering (EMSE)*, 27(17), 2022. doi:10.1007/S10664-021-10057-7. (cited on page 34.)

[GMBG+23] César González-Mora, Cristina Barros, Irene Garrigós, Jose Zubcoff, Elena Lloret, and Jose-Norberto Mazón. Improving open data web API documentation through interactivity and natural language generation. *Computer Standards & Interfaces*, 83(103657), 2023. doi:10.1016/j.csi.2022.103657. (cited on pages 79 and 260.)

[GMWI22] Peter Leo Gorski, Sebastian Möller, Stephan Wiefling, and Luigi Lo Iacono. "I just looked for the solution!"On Integrating Security-Relevant Information in Non-Security API Documentation to Support Secure Coding Practices. *IEEE Transactions on Software Engineering (TSE)*, 48(9):3467–3484, 2022. doi:10.1109/TSE.2021.3094171. (cited on pages 79 and 260.)

[GOSN22] Lisa Geierhaas, Anna-Marie Ortloff, Matthew Smith, and Alena Naiakshina. Let's Hash: Helping Developers with Password Security. In *Proceedings of the 18th Symposium on Usable Privacy and Security (SOUPS)*, pages 503–522, Boston, MA, August 2022. USENIX. URL: https://www.usenix.org/conference/soups2022/presentation/geierhaas. (cited on page 260.)

[Gou13] Georgios Gousios. The GHTorrent Dataset and Tool Suite. In *Proceedings of the 10th International Working Conference on Mining Software Reposi-*

tories (MSR), pages 233–236. IEEE, May 2013. doi:10.1109/MSR.2013.6624034. (cited on page 116.)

[GPKS17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix: Fixing Common C Language Errors by Deep Learning. *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, 31(1), February 2017. doi:10.1609/aaai.v31i1.10742. (cited on page 212.)

[GPT12] Thomas Grill, Ondrej Polacek, and Manfred Tscheligi. Methods towards API Usability: A Structural Analysis of Usability Problem Categories. In *Proceedings of the 4th International Conference on Human-Centered Software Engineering (HCSE)*, pages 164–180. Springer, 2012. doi:10.1007/978-3-642-34347-6_10. (cited on pages 54, 251, and 254.)

[GRS+21] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. Apifix: Output-oriented program synthesis for combating breaking changes in libraries. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), October 2021. doi:10.1145/3485538. (cited on pages 209, 213, and 230.)

[GS10] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 15–24. ACM, 2010. doi:10.1145/1806799.1806806. (cited on pages 96, 99, 101, and 267.)

[GSM+16] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. Automatic Runtime Recovery via Error Handler Synthesis. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 684–695. ACM, 2016. doi:10.1145/2970276.2970360. (cited on page 210.)

[GVES09] Rosalva E. Gallardo-Valencia and Susan Elliott Sim. Internet-Scale Code Search. In *Proceedings of the 1st Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE)*, pages 49–52. IEEE, 2009. doi:10.1109/SUITE.2009.5070022. (cited on page 32.)

[GVIK20] Gao Gao, Finn Voichick, Michelle Ichinco, and Caitlin Kelleher. Exploring Programmers' API Learning Processes: Collecting Web Resources as External Memory. In *Proceedings of the 37th IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10. IEEE, 2020. doi:10.1109/VL/HCC50065.2020.9127274. (cited on pages 54 and 254.)

[GVR02] R.L. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Elsevier Journal of Information and Software Technology (IST)*, 44(8):491–506, 2002. doi:10.1016/S0950-5849(02)00049-6. (cited on page 80.)

[GWL+19] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. An Empirical Study on API-Misuse Bugs in Open-Source C Programs. In *Proceedings of the 43rd Annual International Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 11–20. IEEE, 2019. doi:10.1109/COMPSAC.2019.00012. (cited on pages 27, 54, 97, 253, and 254.)

[GWZ10] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6, 000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–130. ACM, 2010. doi:10.1145/1831708.1831723. (cited on pages 97 and 101.)

[GZHK18] Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. Visualizing API Usage Examples at Scale. In *Proceedings of the 36th Conference on Human Factors in Computing Systems (CHI)*, pages 1–12. ACM, 2018. doi:10.1145/3173574.3174154. (cited on pages 78, 79, 259, and 260.)

[GZK18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep Code Search. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 933–944. ACM, 2018. doi:10.1145/3180155.3180167. (cited on page 32.)

[GZW+15] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. Fixing Recurring Crash Bugs via Analyzing Q&A Sites. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pages 307–318. IEEE/ACM, 2015. doi:10.1109/ASE.2015.81. (cited on pages 213, 230, and 231.)

[HAHH15] Andrew Head, Codanda Appachu, Marti A. Hearst, and Björn Hartmann. Tutorons: Generating Context-Relevant, On-Demand Explanations and Demonstrations of Online Code. In *Proceedings of the 32nd IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 3–12. IEEE, 2015. doi:10.1109/VLHCC.2015.7356972. (cited on pages 78 and 259.)

[HCP+21] Kaifeng Huang, Bihuan Chen, Linghao Pan, Shuai Wu, and Xin Peng. REPFINDER: Finding Replacements for Missing APIs in Library Update. In *Proceedings of the 36th International Conference on Automated Software Engineering (ASE)*, pages 266–278. IEEE, 2021. doi:10.1109/ASE51524.2021.9678905. (cited on pages 35 and 199.)

[HD05] Johannes Henkel and Amer Diwan. CatchUp! capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 274–283. ACM, 2005. doi:10.1145/1062455.1062512. (cited on page 35.)

[HGHH18] Andrew Head, Elena L. Glassman, Björn Hartmann, and Marti A. Hearst. Interactive Extraction of Examples from Existing Code. In *Proceedings of the 36th Conference on Human Factors in Computing Systems*

(CHI), pages 1–12. ACM, 2018. doi:10.1145/3173574.3173659. (cited on pages 79 and 260.)

[HJS+20] Andrew Head, Jason Jiang, James Smith, Marti A. Hearst, and Björn Hartmann. Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Outputs. In *Proceedings of the 38th Conference on Human Factors in Computing Systems (CHI)*, pages 1–12. ACM, 2020. doi:10.1145/3313831.3376798. (cited on page 260.)

[HK99] Kristine Y Hogarty and Jeffrey D Kromrey. Using SAS to calculate tests of Cliff's Delta. In *Proceedings of the SAS Users' Group Int (SUGI)*, pages 1389–1393, 1999. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=c8f0425b4ab5d37980f8bd4394006423c14ac641. (cited on pages 120, 163, 173, 186, and 194.)

[HL11] Daqing Hou and Lin Li. Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC)*, pages 91–100. IEEE, 2011. doi:10.1109/ICPC.2011.21. (cited on pages 4 and 54.)

[HLH+22] Amber Horvath, Michael Xieyang Liu, River Hendriksen, Connor Shannon, Emma Paterson, Kazi Jawad, Andrew Macvean, and Brad A Myers. Understanding How Programmers Can Use Annotations on Documentation. In *Proceedings of the 40th Conference on Human Factors in Computing Systems (CHI)*, pages 1–16. ACM, 2022. doi:10.1145/3491102.3502095. (cited on pages 78, 79, 258, and 260.)

[HLW+18] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in Android apps. In *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*, pages 167–177. ACM/IEEE, 2018. doi:10.1145/3238147.3238185. (cited on pages 35 and 158.)

[HLXX23] Xincheng He, Xiaojin Liu, Lei Xu, and Baowen Xu. How Dynamic Features Affect API Usages? An Empirical Study of API Misuses in Python Programs. In *Proceedings of the 30th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 522–533. IEEE, 2023. doi:10.1109/SANER56733.2023.00055. (cited on pages 54 and 255.)

[HM05] Reid Holmes and Gail C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 117–125. ACM, 2005. doi:10.1145/1062455.1062491. (cited on page 32.)

[HMM23] Amber Horvath, Andrew Macvean, and Brad A. Myers. Support for Long-Form Documentation Authoring and Maintenance. In *Proceedings of the*

40th IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pages 109–114. IEEE, 2023. doi:10.1109/VL-HCC57772.2023.00020. (cited on pages 79 and 261.)

[HNKO20] Robert Heumüller, Sebastian Nielebock, Jacob Krüger, and Frank Ortmeier. Publish or perish, but do not forget your software artifacts. *Springer Empirical Software Engineering (EMSE)*, pages 1–32, 2020. doi:10.1007/s10664-020-09851-6. (cited on pages 97, 98, and 245.)

[HNO18] Robert Heumüller, Sebastian Nielebock, and Frank Ortmeier. Who Plays with Whom? ... And How? Mining API Interaction Patterns from Source Code. In *Proceedings of the 7th International Workshop on Software Mining*, pages 8–11. ACM, 2018. doi:10.1145/3242887.3242888. (cited on page 246.)

[HNO19] Robert Heumüller, Sebastian Nielebock, and Frank Ortmeier. SpecTackle - A Specification Mining Experimentation Platform. In *Proceedings of the 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 178–181. IEEE, 2019. doi:10.1109/SEAA.2019.00036. (cited on page 245.)

[HP14] Jiawei Han and Jian Pei. *Pattern-Growth Methods*, pages 65–81. Springer, Cham, 2014. doi:10.1007/978-3-319-07821-2_3. (cited on pages 36 and 38.)

[HPM+00] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. FreeSpan: Frequent Pattern-projected Sequential Pattern Mining. In Raghu Ramakrishnan, Salvatore J. Stolfo, Roberto J. Bayardo, and Ismail Parsa, editors, *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 355–359. ACM, 2000. doi:10.1145/347090.347167. (cited on page 39.)

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining Frequent Patterns without Candidate Generation. *ACM SIGMOD Record*, 29(2):1–12, May 2000. doi:10.1145/335191.335372. (cited on page 39.)

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, January 1990. doi:10.1145/77606.77608. (cited on page 28.)

[HSSA16] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 237–250. ACM, 2016. doi:10.1145/2908080.2908121. (cited on page 5.)

[Hsu75] Harry T. Hsu. An algorithm for finding a minimal equivalent graph of a digraph. *Journal of the ACM*, 22(1):11–16, 1975. doi:10.1145/321864.321866. (cited on page 150.)

[HTL+21a] Stefanus A. Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller. AndroEvolve: Automated Update for Android Deprecated-API Usages. In *Proceedings of the 43rd Companion of the International Conference on Software Engineering (ICSEC)*, pages 1–4. IEEE, 2021. doi:10.1109/ICSE-Companion52605.2021.00021. (cited on page 35.)

[HTL+21b] Stefanus A. Haryono, Ferdian Thung, David Lo, Julia Lawall, and Lingxiao Jiang. Characterization and Automatic Updates of Deprecated Machine-Learning API Usages. In *Proceedings of the 37th International Conference on Software Maintenance and Evolution (ICSME)*, pages 137–147. IEEE, 2021. doi:10.1109/ICSME52107.2021.00019. (cited on page 35.)

[HW18] Foyzul Hassan and Xiaoyin Wang. HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 1078–1089. ACM, 2018. doi:10.1145/3180155.3180181. (cited on pages 17, 19, and 145.)

[HWL21] Hyunji Hong, Seunghoon Woo, and Heejo Lee. Dicos: Discovering Insecure Code Snippets from Stack Overflow Posts by Leveraging User Discussions. In *Proceedings of the 37th Annual Computer Security Applications Conference*, pages 194–206. ACM, 2021. doi:10.1145/3485832.3488026. (cited on pages 78 and 259.)

[HWM06] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE Transactions on Software Engineering (TSE)*, 32(12):952–970, 2006. doi:10.1109/TSE.2006.117. (cited on pages 78, 79, 259, and 260.)

[HWS20] Ben Hermann, Stefan Winter, and Janet Siegmund. Community Expectations for Research Artifacts and Evaluation Processes. In *Proceedings of the 28th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 469–480. ACM, 2020. doi:10.1145/3368089.3409767. (cited on page 245.)

[HXC+24] Kaifeng Huang, Yingfeng Xia, Bihuan Chen, Siyang He, Huazheng Zeng, Zhuotong Zhou, Jin Guo, and Xin Peng. Your "Notice" Is Missing: Detecting and Fixing Violations of Modification Terms in Open Source Licenses during Forking. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1022–1034. ACM, 2024. doi:10.1145/3650212.3680339. (cited on pages 80 and 261.)

[HXX+18] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. API Method Recommendation without Worrying about the Task-API Knowledge Gap. In *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*, pages 293–304. ACM, 2018. doi:10.1145/3238147.3238191. (cited on pages 32, 78, and 260.)

[HY16] Xue Han and Tingting Yu. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. ACM, 2016. doi:10.1145/2961111.2962602. (cited on page 24.)

[HYM+25] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Transactions on Information Systems (TOIS)*, 43(2), January 2025. doi:10.1145/3703155. (cited on page 247.)

[Ife12] Dirk Ifenthaler. *Measures of Similarity*, pages 2147–2150. Springer, 2012. doi:10.1007/978-1-4419-1428-6_503. (cited on page 152.)

[IHK17] Michelle Ichinco, Wint Yee Hnin, and Caitlin L. Kelleher. Suggesting API Usage to Novice Programmers with the Example Guru. In *Proceedings of the 35th Conference on Human Factors in Computing Systems (CHI)*, pages 1105–1117. ACM, 2017. doi:10.1145/3025453.3025827. (cited on pages 78, 79, 258, 259, and 260.)

[IKND16] Soumya Indela, Mukul Kulkarni, Kartik Nayak, and Tudor Dumitraş. Helping Johnny Encrypt: Toward Semantic Interfaces for Cryptographic Frameworks. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 180–196. ACM, 2016. doi:10.1145/2986012.2986024. (cited on pages 77 and 258.)

[INPR19] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 27th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 510–520. ACM, 2019. doi:10.1145/3338906.3338955. (cited on pages 24 and 54.)

[Ish90] Kaoru Ishikawa. *Introduction to Quality Control*. JUSE Press Ltd., 1990. ISBN-13: 978-94-011-7690-3. (cited on page 4.)

[IWM00] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In *Proceedings of the 4th European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD)*, volume 1910, pages 13–23. Springer, 2000. doi:10.1007/3-540-45372-5_2. (cited on page 40.)

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *Unified Software Development Process*. The Addison-Wesley object technology series. Addison-Wesley, Reading, Mass., 1. printing edition, 1999. ISBN-13: 978-0201571691. (cited on page 4.)

[JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th*

International Conference on Software Engineering (ICSE), pages 467–477. ACM, 2002. doi:10.1145/581339.581397. (cited on page 208.)

[JJE14] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 23rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440. ACM, 2014. doi:10.1145/2610384.2628055. (cited on pages 209 and 213.)

[JM22] Mario Janke and Patrick Mäder. Graph Based Mining of Code Change Patterns From Version Control Commits. *IEEE Transactions on Software Engineering (TSE)*, 48(3):848–863, 2022. doi:10.1109/TSE.2020.3004892. (cited on page 199.)

[JS04] Szymon Jaroszewicz and Dan A. Simovici. Interestingness of Frequent Itemsets using Bayesian Networks as Background Knowledge. In Won Kim, Ron Kohavi, Johannes Gehrke, and William DuMouchel, editors, *Proceedings of the 10th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 178–186. ACM, 2004. doi:10.1145/1014052.1014074. (cited on page 41.)

[JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013. doi:10.1109/ICSE.2013.6606613. (cited on pages 5, 25, 108, and 109.)

[JWL+24] Jiasheng Jiang, Jingzheng Wu, Xiang Ling, Tianyue Luo, Sheng Qu, and Yanjun Wu. APP-Miner: Detecting API Misuses via Automatically Mining API Path Patterns. In *Proceedings of the 45th IEEE Security & Privacy*, pages 4034–4052. IEEE, 2024. doi:10.1109/SP54263.2024.00043. (cited on pages 28, 95, 98, 100, 101, 103, 109, and 267.)

[JXZ+18] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 298–309. ACM, 2018. doi:10.1145/3213846.3213871. (cited on pages 6, 211, and 231.)

[JZW+20] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. An Empirical Study on Bugs Inside TensorFlow. In *Proceedings of the 25th International Conference on Database Systems for Advanced Applications (DASFAA)*, volume 12112, pages 604–620. Springer, 2020. doi:10.1007/978-3-030-59410-7_40. (cited on pages 24 and 54.)

[KAB20] Stefan Krüger, Karim Ali, and Eric Bodden. CogniCryptGEN: generating code for the secure usage of crypto APIs. In *Proceedings of the 18th International Symposium on Code Generation and Optimization (CGO)*, pages 185–198. ACM, 2020. doi:10.1145/3368826.3377905. (cited on pages 78 and 259.)

[Kan06] Gopal K. Kanji. *100 Statistical Tests*. SAGE, 3 edition, 2006. ISBN-13: 978-1847878267. (cited on pages 120, 125, 163, 173, 186, and 194.)

[KAS13] Vasanth Krishnamoorthy, Bharatwaj Appasamy, and Christopher Scaffidi. Using Intelligent Tutors to Teach Students How APIs Are Used for Software Engineering in Practice. *IEEE Transactions on Education*, 56(3):355–363, 2013. doi:10.1109/TE.2013.2238543. (cited on pages 78 and 259.)

[KB23] Caitlin Kelleher and Michelle Brachman. A sensemaking analysis of API learning using React. *Journal of Computer Languages (COLA)*, 74(101189), 2023. doi:10.1016/j.cola.2022.101189. (cited on pages 54 and 251.)

[KBK+17] Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Impact of Tool Support in Patch Construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 237–248. ACM, 2017. doi:10.1145/3092703.3092713. (cited on pages 207 and 247.)

[KC07] Babara Ann Kitchenham and Stuart Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report 2.3, Software Engineering Group, Keele University; Department of Computer Science, University of Durham, July 2007. EBSE 2007-001, Joint Report. URL: https://www.researchgate.net/publication/302924724_Guidelines_for_performing_Systematic_Literature_Reviews_in_Software_Engineering. (cited on pages 7, 12, 15, 45, 48, 49, 69, 88, 91, 93, and 107.)

[KCJ22] Chia Hung Kao, Cheng-Ying Chang, and Hewijin Christine Jiau. Towards cost-effective API deprecation: A win–win strategy for API developers and API users. *Elsevier Journal of Information and Software Technology (IST)*, 142:106746, 2022. doi:10.1016/j.infsof.2021.106746. (cited on page 35.)

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007. doi:10.1002/smr.344. (cited on pages 33 and 34.)

[KCT12] Marcos Kalinowski, David N. Card, and Guilherme H. Travassos. Evidence-Based Guidelines to Defect Causal Analysis. *IEEE Software*, 29(4):16–18, 2012. doi:10.1109/MS.2012.72. (cited on page 4.)

[KFLS18] Maria Kechagia, Marios Fragkoulis, Panos Louridas, and Diomidis Spinellis. The exception handling riddle: An empirical study on the Android API. *Elsevier Journal of Systems and Software (JSS)*, 142:248–270, 2018. doi:10.1016/j.jss.2018.04.034. (cited on page 54.)

[KGH+17] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A Critical Evaluation of Spectrum-Based Fault Localization Techniques on a Large-Scale Software System. In *Proceedings of the 17th International Conference on Software Quality, Reliability and Security (QRS)*, pages 114–125. IEEE, 2017. doi:10.1109/QRS.2017.22. (cited on page 208.)

[KK01] Michihiro Kuramochi and George Karypis. Frequent Subgraph Discovery. In *Proceedings of the 1st International Conference on Data Mining (ICDM)*, pages 313–320. IEEE, 2001. doi:10.1109/ICDM.2001.989534. (cited on page 40.)

[KL21] Hong Jin Kang and David Lo. Active Learning of Discriminative Subgraph Patterns for API Misuse Detection. *IEEE Transactions on Software Engineering (TSE)*, pages 1–1, 2021. doi:10.1109/TSE.2021.3069978. (cited on pages 5, 27, 31, 96, 97, 99, 101, 103, 104, 115, 133, 136, 137, 139, 157, 163, 173, 179, 196, 197, 198, 200, 202, 220, 240, 244, 246, and 267.)

[Kle98] Jon M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 668–677. SIAM, 1998. URL: https://cse.msu.edu/~cse960/Papers/LinkAnalysis/auth.pdf. (cited on page 154.)

[KLHK13] Jinhan Kim, Sanghoon Lee, Seung-Won Hwang, and Sunghun Kim. Enriching Documents with Examples: A Corpus Mining Approach. *Transactions on Information Systems*, 31(1), January 2013. doi:10.1145/2414782.2414783. (cited on pages 79 and 260.)

[KLvN+20] Jacob Krüger, Christian Lausberger, Ivonne von Nostitz-Wallwitz, Gunter Saake, and Thomas Leich. Search. Review. Repeat? An empirical study of threats to replicating SLR searches. *Springer Empirical Software Engineering (EMSE)*, 25(1):627–677, 2020. doi:10.1007/S10664-019-09763-0. (cited on pages 48, 49, 68, 71, 85, 93, and 107.)

[KMB+17] Barbara A Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart M Charters, Shirley Gibbs, and Amnart Pohthong. Robust Statistical Methods for Empirical Software Engineering. *Springer Empirical Software Engineering (EMSE)*, 22(2):579–630, 2017. doi:10.1007/s10664-016-9437-5. (cited on pages 120, 163, 173, 186, and 194.)

[KMCR12] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: On-Demand Infinite Loop Escape in Unmodified Binaries. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 431–450. ACM, 2012. doi:10.1145/2384616.2384648. (cited on page 210.)

[KMS14] Maria Kechagia, Dimitris Mitropoulos, and Diomidis Spinellis. Charting the API minefield using software telemetry data. *Springer Empirical Software Engineering (EMSE)*, 20(6):1785–1830, December 2014. doi:10.1007/S10664-014-9343-7. (cited on pages 54, 251, and 253.)

[KMSH21] Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman. Evaluating Automatic Program Repair Capabilities to Repair API Misuses. *IEEE Transactions on Software Engineering (TSE)*, pages 1–1, 2021. doi:10.1109/TSE.2021.3067156. (cited on pages 3, 6, 18, 139, 205, 206, 209, 210, 211, 212, 220, 222, 231, 236, 243, and 246.)

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Mass., third edition, 1997. ISBN-13: 978-0-201-89683-1. (cited on page 233.)

[KPW06] Sunghun Kim, Kai Pan, and E. E. James Whitehead. Memories of Bug Fixes. In *Proceedings of the 5th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 35–45. ACM, 2006. doi:10.1145/1181775.1181781. (cited on pages 145, 213, 230, and 231.)

[KRW+23] Stefan Krüger, Michael Reif, Anna-Katharina Wickert, Sarah Nadi, Karim Ali, Eric Bodden, Yasemin Acar, Mira Mezini, and Sascha Fahl. Securing Your Crypto-API Usage Through Tool Support - A Usability Study. In *2023 IEEE Secure Development Conference (SecDev)*, pages 14–25. IEEE, 2023. doi:10.1109/SecDev56634.2023.00015. (cited on pages 78, 79, 258, and 259.)

[KSA+21] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. *IEEE Transactions on Software Engineering (TSE)*, 47(11):2382–2400, 2021. doi:10.1109/TSE.2019.2948910. (cited on pages 2, 5, 96, 98, 99, 100, 101, 105, 198, 200, and 267.)

[KSK+24] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. SootUp: A Redesign of the Soot Static Analysis Framework. In Bernd Finkbeiner and Laura Kovács, editors, *Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 229–247, Cham, 2024. Springer. doi:10.1007/978-3-031-57246-3_13. (cited on page 25.)

[KWM+21] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, Tracy Hall, Saemundur Haraldsson, and John Woodward. On The Introduction of Automatic Program Repair in Bloomberg. *IEEE Software*, 38(4):43–51, 2021. doi:10.1109/MS.2021.3071086. (cited on pages 18, 207, 208, 215, 231, and 247.)

[LAH18] Tianshi Li, Yuvraj Agarwal, and Jason I. Hong. Coconut: An ide plugin for developing privacy-friendly apps. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(4), December 2018. doi:10.1145/3287056. (cited on pages 78 and 259.)

[LBWK18] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. CiD: Automating the detection of API-related compatibility issues in android

apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 153–163. ACM, 2018. doi:10.1145/3213846.3213857. (cited on page 158.)

[LCC+23] Ke Liu, Xiang Chen, Chunyang Chen, Xiaofei Xie, and Zhanqi Cui. Automated Question Title Reformulation by Mining Modification Logs From Stack Overflow. *IEEE Transactions on Software Engineering (TSE)*, 49(9):4390–4410, 2023. doi:10.1109/TSE.2023.3292399. (cited on pages 78 and 259.)

[LCP+21] Wenjian Liu, Bihuan Chen, Xin Peng, Qinghao Sun, and Wenyun Zhao. Identifying change patterns of API misuses from code changes. *Science China Information Sciences*, 64(3), 2021. doi:10.1007/S11432-019-2745-5. (cited on pages 5, 95, 98, 99, 100, 101, 102, 105, 198, 200, and 267.)

[Leu09] Carson Kai-Sang Leung. *Anti-monotone Constraints*, pages 98–98. Springer, 2009. doi:10.1007/978-0-387-39940-9_5046. (cited on page 38.)

[LGC+12] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic Input Rectification. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 80–90. IEEE, 2012. doi:10.1109/ICSE.2012.6227204. (cited on page 210.)

[LGDVFW12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012. doi:10.1109/ICSE.2012.6227211. (cited on page 211.)

[LGFW13] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in Automatic Software Repair. *Software Quality Journal*, 21:421–443, 2013. doi:10.1007/s11219-013-9208-0. (cited on pages 6, 205, 206, 207, and 208.)

[LGPR19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated Program Repair. *Communications of the ACM*, 62(12):56–65, November 2019. doi:10.1145/3318162. (cited on pages 3, 6, 18, 24, 114, 205, 206, 207, 208, 211, and 220.)

[LGS21] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A Systematic Review of API Evolution Literature. *ACM Computing Surveys*, 54(8), October 2021. doi:10.1145/3470133. (cited on pages 34, 46, 47, 70, and 102.)

[LHT+22] Kien Luong, Mohammad Hadi, Ferdian Thung, Fatemeh Fard, and David Lo. ARSeek: Identifying API Resource using Code and Discussion on Stack Overflow. In *Proceedings of the 30th International Conference on Program Comprehension (ICPC)*, pages 331–342. ACM, 2022. doi:10.1145/3524610.3527918. (cited on pages 78 and 259.)

[LHX+16] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. How Good Are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 602–613. ACM, 2016. doi:10.1145/2970276.2970356. (cited on pages 5 and 143.)

[Lin15] Christian Lindig. Mining Patterns and Violations Using Concept Analysis. In Christian Bird, Tim Menzies, and Thomas Zimmermann, editors, *The Art and Science of Analyzing Software Data*, pages 17–38. Morgan Kaufmann / Elsevier, 2015. doi:10.1016/B978-0-12-411519-4.00002-1. (cited on page 2.)

[LJB+21] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. A Large-scale Study on API Misuses in the Wild. In *Proceedings of the 14th International Conference on Software Testing, Verification and Validation (ICST)*, pages 241–252. IEEE, 2021. doi:10.1109/ICST49551.2021.00034. (cited on pages 26 and 27.)

[LK77] J. Richard Landis and Gary G. Koch. The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159–174, 1977. doi:10.2307/2529310. (cited on page 149.)

[LKKB19a] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 31–42. ACM, 2019. doi:10.1145/3293882.3330577. (cited on pages 211 and 231.)

[LKKB19b] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyandè. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 1–12. IEEE, 2019. doi:10.1109/SANER.2019.8667970. (cited on pages 211 and 231.)

[LL15] Tien-Duy B. Le and David Lo. Beyond Support and Confidence: Exploring Interestingness Measures for Rule-Based Specification Mining. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 331–340. IEEE, 2015. doi:10.1109/SANER.2015.7081843. (cited on pages 36, 40, and 109.)

[LLLG16] Xuan Bach D Le, David Lo, and Claire Le Goues. History Driven Program Repair. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016. doi:10.1109/SANER.2016.76. (cited on pages 6, 17, 19, and 145.)

[LLS+18] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. Improving API Caveats Accessibility by Mining API Caveats Knowledge Graph. In *Proceedings of the 34th International*

Conference on Software Maintenance and Evolution (ICSME), pages 183–193. IEEE, 2018. doi:10.1109/ICSME.2018.00028. (cited on pages 78, 79, 258, and 260.)

[LMC+21] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. ARBITRAR: User-Guided API Misuse Detection. In *Proceedings of the 42nd IEEE Security & Privacy*, pages 1400–1415. IEEE, 2021. doi:10.1109/SP40001.2021.00090. (cited on pages 2, 96, 99, 100, 101, 103, and 267.)

[LP17] Frank Li and Vern Paxson. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 2201–2215. ACM, 2017. doi:10.1145/3133956.3134072. (cited on page 24.)

[LPM+19] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. Generating Query-Specific Class API Summaries. In *Proceedings of the 27th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 120–130. ACM, 2019. doi:10.1145/3338906.3338971. (cited on page 259.)

[LPM+21] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Xuefang Bai, Gang Lyu, Jiazhan Xie, and Xiaoxin Zhang. Learning-Based Extraction of First-Order Logic Representations of API Directives. In *Proceedings of the 29th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 491–502. ACM, 2021. doi:10.1145/3468264.3468618. (cited on pages 78 and 259.)

[LPP+20] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 101–114. ACM, 2020. doi:10.1145/3395363.3397369. (cited on page 212.)

[LRK+19] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys*, 52(6), November 2019. doi:10.1145/3359981. (cited on page 4.)

[LS18] Maxime Lamothe and Weiyi Shang. Exploring the use of automated API migrating techniques in practice: an experience report on Android. In *Proceedings of the 15th International Working Conference on Mining Software Repositories (MSR)*, pages 503–514. ACM, 2018. doi:10.1145/3196398.3196420. (cited on pages 17 and 35.)

[LS20] Maxime Lamothe and Weiyi Shang. When APIs Are Intentionally Bypassed: An Exploratory Study of API Workarounds. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages

912–924. ACM, 2020. doi:10.1145/3377811.3380433. (cited on pages 4, 54, and 254.)

[LSC22] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. A3: Assisting Android API Migrations Using Code Examples. *IEEE Transactions on Software Engineering (TSE)*, 48(2):417–431, 2022. doi:10.1109/TSE.2020.2988396. (cited on pages 35, 79, 199, and 261.)

[LSDR14] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic Runtime Error Repair and Containment via Recovery Shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 227–238. ACM, 2014. doi:10.1145/2594291.2594337. (cited on page 210.)

[LTLLG18] Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in Semantics-based Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, page 163. ACM, 2018. doi:10.1145/3180155.3182536. (cited on pages 208 and 246.)

[LTW+06] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have Things Changed Now? - An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 25–33. ACM, 2006. doi:10.1145/1181309.1181314. (cited on page 24.)

[LVBDP+14] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do API changes trigger stack overflow discussions? a study on the Android SDK. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, pages 83–94. ACM, 2014. doi:10.1145/2597008.2597155. (cited on pages 54 and 253.)

[LW09] Claire Le Goues and Westley Weimer. Specification Mining with Few False Positives. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, pages 292–306. Springer, 2009. doi:10.1007/978-3-642-00768-2_26. (cited on pages 2, 3, 36, and 103.)

[LW12] Claire Le Goues and Westley Weimer. Measuring Code Quality to Improve Specification Mining. *IEEE Transactions on Software Engineering (TSE)*, 38(1):175–190, 2012. doi:10.1109/TSE.2011.5. (cited on pages 14 and 109.)

[LWCK21] Seonah Lee, Rongxin Wu, Shing-Chi Cheung, and Sungwon Kang. Automatic Detection and Update Suggestion for Outdated API Names in Documentation. *IEEE Transactions on Software Engineering (TSE)*, 47(4):653–675, 2021. doi:10.1109/TSE.2019.2901459. (cited on pages 79 and 261.)

[LWL+22] Zhenming Li, Ying Wang, Zeqi Lin, Shing-Chi Cheung, and Jian-Guang Lou. Nufix: Escape From NuGet Dependency Maze. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 1545–1557. ACM, 2022. doi:10.1145/3510003.3510118. (cited on pages 79 and 260.)

[LXL+21] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. Opportunities and Challenges in Code Search Tools. *ACM Computing Surveys*, 54(9), October 2021. doi:10.1145/3480027. (cited on page 32.)

[LYY+23] Junwei Luo, Xuechao Yang, Xun Yi, Fengling Han, Iqbal Gondal, and Guang-Bin Huang. A Comparative Study on Design and Usability of Cryptographic Libraries. In *Proceedings of the 9th Australasian Computer Science Week (ACSW)*, pages 102–111. ACM, 2023. doi:10.1145/3579375.3579388. (cited on pages 54, 250, and 253.)

[LZ05] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 4th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 306–315. ACM, 2005. doi:10.1145/1081706.1081755. (cited on pages 5, 28, 95, 101, 109, and 267.)

[LZL+15] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pages 260–270. IEEE, 2015. doi:10.1109/ASE.2015.42. (cited on page 32.)

[LZP+23] Mingwei Liu, Chengyuan Zhao, Xin Peng, Simin Yu, Haofen Wang, and Chaofeng Sha. Task-Oriented ML/DL Library Recommendation Based on a Knowledge Graph. *IEEE Transactions on Software Engineering (TSE)*, 49(8):4081–4096, 2023. doi:10.1109/TSE.2023.3285280. (cited on pages 32, 78, 258, and 260.)

[LZT+24] Can Li, Jingxuan Zhang, Yixuan Tang, Zhuhang Li, and Tianyue Sun. Boosting API Misuse Detection via Integrating API Constraints from Multiple Sources. In *Proceedings of the 21st International Working Conference on Mining Software Repositories (MSR)*, pages 14–26. ACM, 2024. doi:10.1145/3643991.3644904. (cited on pages 96, 98, 99, 101, 106, 198, 200, and 267.)

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003. ISBN-13: 978-0135974445. (cited on page 4.)

[Mar13] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*, volume 12. Pearson Education Inc., 2013. ISBN-13: 978-0-13-235088-4. (cited on page 4.)

[MBC+19] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. SapFix: Automated End-to-End Repair at Scale. In *Proceedings of the 41st International Conference on Software Engineering - Software Engineering in Practice (ICSE-SEIP)*, pages 269–278. IEEE, 2019. doi:10.1109/ICSE-SEIP.2019.00039. (cited on pages 207, 208, 215, 231, 234, and 247.)

[MBDP+15] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How Can I Use This Method? In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, volume 1, pages 880–890. IEEE, 2015. doi:10.1109/ICSE.2015.98. (cited on page 32.)

[MBM10] Martin Monperrus, Marcel Bruch, and Mira Mezini. Detecting Missing Method Calls in Object-Oriented Software. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*, volume 6183, pages 2–25. Springer, 2010. doi:10.1007/978-3-642-14107-2_2. (cited on pages 2, 93, 95, 99, 101, 102, and 267.)

[MBP+17] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. ARENA: An Approach for the Automated Generation of Release Notes. *IEEE Transactions on Software Engineering (TSE)*, 43(2):106–127, 2017. doi:10.1109/TSE.2016.2591536. (cited on page 35.)

[MBS+19] Ariana Mirian, Nikunj Bhagat, Caitlin Sadowski, Adrienne Porter Felt, Stefan Savage, and Geoffrey M. Voelker. Web Feature Deprecation: A Case Study for Chrome. In *Proceedings of the 41st International Conference on Software Engineering - Software Engineering in Practice (ICSE-SEIP)*, pages 302–311. IEEE, 2019. doi:10.1109/ICSE-SEIP.2019.00044. (cited on page 34.)

[McG23] Leon McGregor. *Games and interactions to motivate the secure and analytical mindsets of developers*. PhD thesis, Heriot-Watt University, School of Mathematical and Computer Sciences, 2023. URL: http://hdl.handle.net/10399/4975. (cited on pages 46, 47, 70, and 71.)

[MCJ17] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding api usage errors. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference/-Foundations of Software Engineering (ESEC/FSE)*, pages 151–162. ACM, 2017. doi:10.1145/3106237.3106284. (cited on pages 2, 5, 96, 98, 100, 101, 107, 198, 200, and 267.)

[MCP98] Florent Masseglia, Fabienne Cathala, and Pascal Poncelet. The PSP Approach for Mining Sequential Patterns. In Jan M. Zytkow and Mohamed Quafafou, editors, *Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD)*, volume 1510, pages 176–184. Springer, 1998. doi:10.1007/BFb0094818. (cited on page 39.)

[METM12] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? an empirical study on the directives of API documentation. *Springer Empirical Software Engineering (EMSE)*, 17(6):703–737, 2012. doi:10.1007/S10664-011-9186-4. (cited on page 26.)

[MGP+11] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding Relevant Functions and Their Usage. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 111–120. ACM, 2011. doi:10.1145/1985793.1985809. (cited on page 32.)

[MHR+12] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 683–702. ACM, 2012. doi:10.1145/2384616.2384666. (cited on pages 80 and 261.)

[MJHS90] R. G. Mays, C. L. Jones, G. J. Holloway, and D. P. Studinski. Experiences with Defect Prevention. *IBM Systems Journal*, 29(1):4–32, 1990. doi:10.1147/sj.291.0004. (cited on pages 3 and 14.)

[MKA+18] Lauren Murphy, Mary Beth Kery, Oluwatosin Alliyu, Andrew Macvean, and Brad A. Myers. API Designers in the Field: Design Practices and Challenges for Creating Usable APIs. In *Proceedings of the 35th IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 249–258. IEEE, 2018. doi:10.1109/VLHCC.2018.8506523. (cited on pages 2, 4, 54, 59, and 250.)

[ML23] Sahar Mehrpour and Thomas D. LaToza. Can static analysis tools find more defects? *Springer Empirical Software Engineering (EMSE)*, 28(5), 2023. doi:10.1007/S10664-022-10232-4. (cited on page 25.)

[MLK19] Sahar Mehrpour, Thomas D. LaToza, and Rahul K. Kindi. Active Documentation: Helping Developers Follow Design Decisions. In *Proceedings of the 36th IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 87–96. IEEE, 2019. doi:10.1109/VLHCC.2019.8818816. (cited on pages 79 and 261.)

[MLLD16] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 711–722. ACM, 2016. doi:10.1145/2897845.2897896. (cited on pages 209, 213, 230, and 231.)

[MM16a] Matias Martinez and Martin Monperrus. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 441–444. ACM, 2016. doi:10.1145/2931037.2948705. (cited on page 211.)

[MM16b] Paul W. McBurney and Collin McMillan. Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering (TSE)*, 42(2):103–119, 2016. doi:10.1109/TSE.2015.2465386. (cited on pages 79 and 260.)

[MM18] Matias Martinez and Martin Monperrus. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *Proceedings of the 10th International ymposium on Search-Based Software Engineering SSBSE*, volume 11036, pages 65–86. Springer, 2018. doi:10.1007/978-3-319-99241-9_3. (cited on page 211.)

[MNY+18] Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo Arango Argoty. Secure Coding Practices in Java: Challenges and Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 372–383. ACM, 2018. doi:10.1145/3180155.3180201. (cited on pages 54, 100, 251, and 253.)

[Mon18a] Martin Monperrus. Automatic Software Repair: A Bibliography. *ACM Computing Surveys*, 51(1):17:1–24, 2018. doi:10.1145/3105906. (cited on pages 3, 6, 18, 205, 206, 207, 209, 210, and 236.)

[Mon18b] Martin Monperrus. The Living Review on Automated Program Repair. Technical Report hal-01956501, HAL Archives Ouvertes, 2018. URL: https://hal.science/hal-01956501. (cited on pages 205, 206, 209, and 236.)

[MPCOF+21] Fernando Martínez-Plumed, Lidia Contreras-Ochando, Cèsar Ferri, José Hernández-Orallo, Meelis Kull, Nicolas Lachiche, María José Ramírez-Quintana, and Peter Flach. CRISP-DM Twenty Years Later: From Data Mining Processes to Data Science Trajectories. *IEEE Transactions on Knowledge and Data Engineering*, 33(8):3048–3061, 2021. doi:10.1109/TKDE.2019.2962680. (cited on pages 41 and 42.)

[MRARMB+18] Eduardo Mosqueira-Rey, David Alonso-Ríos, Vicente Moret-Bonillo, Isaac Fernández-Varela, and Diego Álvarez Estévez. A systematic approach to API usability: Taxonomy-derived criteria and a case study. *Elsevier Journal of Information and Software Technology (IST)*, 97:46–63, 2018. doi:10.1016/j.infsof.2017.12.010. (cited on pages 54, 78, 253, and 258.)

[MRF19] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education*, 62(2):77–90, 2019. doi:10.1109/TE.2018.2864133. (cited on page 4.)

[MRK13] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 29th International Conference on Software Maintenance (ICSM)*, pages 70–79. IEEE, 2013. doi:10.1109/ICSM.2013.18. (cited on pages 54 and 252.)

[MSS18] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication (JTWC)*, 48(3):295–330, 2018. doi:10.1177/0047281617721853. (cited on pages 4, 54, 251, 252, and 254.)

[MSS20] Michael Meng, Stephanie M. Steinhardt, and Andreas Schubert. Optimizing API Documentation: Some Guidelines and Effects. In *Proceedings of the 38th International Conference on Design of Communication (SIGDOC)*, pages 1–11. ACM, 2020. doi:10.1145/3380851.3416759. (cited on pages 79 and 260.)

[MT97] Heikki Mannila and Hannu Toivonen. Levelwise Search and Borders of Theories in Knowledge Discovery. *Springer Data Mining and Knowledge Discovery*, 1(3):241–258, 1997. doi:10.1023/A:1009796218281. (cited on page 37.)

[MTV94] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient Algorithms for Discovering Association Rules. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 181–192. AAAI, 1994. URL: https://cdn.aaai.org/Workshops/1994/WS-94-03/WS94-03-016.pdf. (cited on page 38.)

[MUKS24] Iren Mazloomzadeh, Gias Uddin, Foutse Khomh, and Ashkan Sami. Reputation Gaming in Crowd Technical Knowledge Sharing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 34(1), December 2024. doi:10.1145/3691627. (cited on pages 78 and 259.)

[MUMM19] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 468–478. IEEE, 2019. doi:10.1109/SANER.2019.8667991. (cited on pages 209 and 220.)

[Mun57] James Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1), 1957. URL: https://www.jstor.org/stable/2098689. (cited on pages 149, 154, 217, 221, and 222.)

[MUR23] Saikat Mondal, Gias Uddin, and Chanchal K. Roy. Automatic prediction of rejected edits in Stack Overflow. *Springer Empirical Software Engineering (EMSE)*, 28(9), 2023. doi:10.1007/S10664-022-10242-2. (cited on page 259.)

[MWD24] May Mahmoud, Robert J. Walker, and Jörg Denzinger. API usage templates via structural generalization. *Elsevier Journal of Systems and Software (JSS)*, 210(111974), 2024. doi:10.1016/j.jss.2024.111974. (cited on pages 78, 258, and 259.)

[MWZM12] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 353–363. IEEE, 2012. doi:10.1109/ICSE.2012.6227179. (cited on pages 17, 35, and 199.)

[MXBK05] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 48–61. ACM, 2005. doi:10.1145/1065010.1065018. (cited on pages 32, 78, 79, 259, and 260.)

[MYR16] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 691–701. ACM, 2016. doi:10.1145/2884781.2884807. (cited on pages 211 and 231.)

[NAP18] Kristian Nybom, Adnan Ashraf, and Ivan Porres. A Systematic Mapping Study on API Documentation Generation Approaches. In *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 462–469. IEEE, 2018. doi:10.1109/SEAA.2018.00081. (cited on pages 4, 46, 47, 70, and 71.)

[NBKO21a] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. An Experimental Analysis of Graph-Distance Algorithms for Comparing API Usages. In *Proceedings of the 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 214–225. IEEE, 2021. doi:10.1109/SCAM52516.2021.00034. (cited on pages 15, 17, 143, 149, 152, 154, 155, and 156.)

[NBKO21b] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. AndroidCompass: A Dataset of Android Compatibility Checks in Code Repositories. In *Proceedings of the 18th International Working Conference on Mining Software Repositories (MSR)*, pages 535–539. IEEE, 2021. doi:10.1109/MSR52588.2021.00069. (cited on pages 139, 157, 158, 196, 197, 202, 236, and 244.)

[NBKO22] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. Automated Change Rule Inference for Distance-Based API Misuse Detection, 2022. doi:10.48550/ARXIV.2207.06665. (cited on pages 15, 17, 91, 143, 149, 155, 156, 270, and 271.)

[NBKO24] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. ASAP-Repair: API-Specific Automated Program Repair Based on API Usage Graphs. In *Proceedings of the 5th International Workshop on Automated Program Repair (APR)*, pages 1–4. ACM, 2024. doi:10.1145/3643788.3648011. (cited on pages 205 and 224.)

[NDBB20] Duc Cuong Nguyen, Erik Derr, Michael Backes, and Sven Bugiel. Up2Dep: Android Tool Support to Fix Insecure Code Dependencies. In *Proceedings*

of the 36th Annual Computer Security Applications Conference (ACSAC), pages 263–276. ACM, 2020. doi:10.1145/3427228.3427658. (cited on pages 79 and 261.)

[NDRDS+22] Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Davide Di Ruscio, and Massimiliano Di Penta. Recommending API Function Calls and Code Snippets to Support Software Development. *IEEE Transactions on Software Engineering (TSE)*, 48(7):2417–2438, 2022. doi:10.1109/TSE.2021.3059907. (cited on pages 78 and 259.)

[NDT+17] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why Do Developers Get Password Storage Wrong? A Qualitative Usability Study. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 311–328. ACM, 2017. doi:10.1145/3133956.3134082. (cited on pages 54, 59, 78, 250, 251, 255, and 258.)

[NGB21] AmirHossein Naghshzan, Latifa Guerrouj, and Olga Baysal. Leveraging Unsupervised Learning to Summarize APIs Discussed in Stack Overflow. In *Proceedings of the 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 142–152. IEEE, 2021. doi:10.1109/SCAM52516.2021.00026. (cited on pages 78 and 259.)

[NHC+16] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. API Code Recommendation using Statistical Learning from Fine-Grained Changes. In *Proceedings of the 24th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 511–522. ACM, 2016. doi:10.1145/2950290.2950333. (cited on page 199.)

[NHKO20a] Sebastian Nielebock, Robert Heumüller, Jacob Krüger, and Frank Ortmeier. Cooperative API Misuse Detection Using Correction Rules. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE) - New Ideas and Emerging Results Track*, pages 73–76. ACM, 2020. doi:10.1145/3377816.3381735. (cited on pages 15, 17, and 143.)

[NHKO20b] Sebastian Nielebock, Robert Heumüller, Jacob Krüger, and Frank Ortmeier. Using API-Embedding for API-Misuse Repair. In *Proceedings of the 1st International Workshop on Automated Program Repair (APR)*, pages 1–2. ACM, 2020. doi:10.1145/3387940.339217. (cited on page 248.)

[NHL18] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. CLEVER: Combining Code Metrics with Clone Detection for Just-in-Time Fault Prevention and Resolution in Large Industrial Projects. In *Proceedings of the 15th International Working Conference on Mining Software Repositories (MSR)*, pages 153–164. ACM, 2018. doi:10.1145/3196398.3196438. (cited on pages 18, 207, 215, 231, and 247.)

[NHM+19] Daye Nam, Amber Horvath, Andrew Macvean, Brad Myers, and Bogdan Vasilescu. MARBLE: Mining for Boilerplate Code to Identify API

Usability Problems. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE)*, pages 615–627. IEEE, 2019. doi:10.1109/ASE.2019.00063. (cited on pages 54, 251, and 252.)

[NHO18] Sebastian Nielebock, Robert Heumüller, and Frank Ortmeier. Commits as a Basis for API Misuse Detection. In *Proceedings of the 7th International Workshop on Software Mining*, pages 20–23. ACM, 2018. doi:10.1145/3242887.3242890. (cited on pages 14, 16, 91, 108, 111, and 247.)

[NHSO21] Sebastian Nielebock, Robert Heumüller, Kevin Michael Schott, and Frank Ortmeier. Guided Pattern Mining for API Misuse Detection by Change-Based Code Analysis. *Springer Automated Software Engineering*, 28(15), August 2021. doi:10.1007/s10515-021-00294-x. (cited on pages 14, 16, 91, 108, 109, 110, 111, 113, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 130, 132, 134, 135, and 137.)

[Nie17] Sebastian Nielebock. Towards API-Specific Automatic Program Repair. In *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE) - Doctoral Symposium*, pages 1010–1013. IEEE, 2017. doi:10.1109/ASE.2017.8115721. (cited on pages 6, 19, 205, 207, 209, and 210.)

[NK11] Anh Cuong Nguyen and Siau-Cheng Khoo. Extracting Significant Specifications from Mining through Mutation Testing. In *Proceedings of the 13th International Conference on Formal Engineering Methods (ICFEM)*, volume 6991, pages 472–488. Springer, 2011. doi:10.1007/978-3-642-24559-6_32. (cited on pages 2, 5, 95, 101, and 267.)

[NKMB16] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through Hoops: Why Do Java Developers Struggle with Cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 935–946. ACM, 2016. doi:10.1145/2884781.2884790. (cited on pages 4, 54, 249, and 250.)

[NKZ17] Haoran Niu, Iman Keivanloo, and Ying Zou. Learning to rank code examples for code search engines. *Springer Empirical Software Engineering (EMSE)*, 22(1):259–291, 2017. doi:10.1007/S10664-015-9421-5. (cited on pages 78, 79, 259, and 260.)

[NL22] Amirfarhad Nilizadeh and Gary T. Leavens. Be Realistic: Automated Program Repair is a Combination of Undecidable Problems. In *Proceedings of the 3rd International Workshop on Automated Program Repair (APR)*, pages 31–32. ACM, 2022. doi:10.1145/3524459.3527346. (cited on pages 6, 18, and 207.)

[NM10] Seyed Mehdi Nasehi and Frank Maurer. Unit tests as API usage examples. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE, 2010. doi:10.1109/ICSM.2010.5609553. (cited on page 259.)

[NMH+24] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an LLM to Help With Code Understanding. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. ACM, 2024. doi:10.1145/3597503.3639187. (cited on pages 78, 79, 247, and 258.)

[NMR22] Md. Nadim, Debajyoti Mondal, and Chanchal K. Roy. Leveraging structural properties of source code graphs for just-in-time bug prediction. *Springer Automated Software Engineering*, 29(1):27, 2022. doi:10.1007/S10515-022-00326-0. (cited on page 199.)

[NMVH23] Daye Nam, Brad Myers, Bogdan Vasilescu, and Vincent Hellendoorn. Improving API Knowledge Discovery with ML: A Case Study of Comparable API Methods. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pages 1890–1906. IEEE, 2023. doi:10.1109/ICSE48619.2023.00161. (cited on pages 79 and 260.)

[NND+19] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-Based Mining of In-the-Wild, Fine-Grained, Semantic Code Change Patterns. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 819–830, 2019. doi:10.1109/ICSE.2019.00089. (cited on page 199.)

[NNP+09a] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 440–455. Springer, 2009. doi:10.1007/978-3-642-00593-0_31. (cited on pages 43, 106, 155, 202, and 243.)

[NNP+09b] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 383–392. ACM, 2009. doi:10.1145/1595696.1595767. (cited on pages 2, 5, 28, 93, 95, 101, 102, 105, and 267.)

[NNPN17] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring API Embedding for API Usages and Applications. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 438–449. IEEE, 2017. doi:10.1109/ICSE.2017.47. (cited on pages 32, 35, 199, and 248.)

[NNW+10] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. *SIGPLAN Not.*, 45(10):302–321, October 2010. doi:10.1145/1932682.1869486. (cited on pages 35 and 199.)

[NR25] Mathieu Nassif and Martin P. Robillard. Non-Linear Software Documentation with Interactive Code Examples. *ACM Transactions on*

Software Engineering and Methodology (TOSEM), 34(2), January 2025. doi:10.1145/3702976. (cited on pages 79 and 261.)

[NVN19] Tam The Nguyen, Phong Minh Vu, and Tung Thanh Nguyen. An Empirical Study of Exception Handling Bugs and Fixes. In *Proceedings of the 2019 ACM Southeast Conference (ACSME)*, pages 257–260. ACM, 2019. doi:10.1145/3299815.3314472. (cited on page 24.)

[NVN20] Tam Nguyen, Phong Vu, and Tung Nguyen. Code Recommendation for Exception Handling. In *Proceedings of the 28th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 1027–1038. ACM, 2020. doi:10.1145/3368089.3409690. (cited on pages 5, 95, 100, 101, 102, 105, 198, 200, and 267.)

[NWA+17] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 1065–1077. ACM, 2017. doi:10.1145/3133956.3133977. (cited on pages 78, 258, and 259.)

[NZS+23] Dip Kiran Pradhan Newar, Rui Zhao, Harvey Siy, Leen-Kiat Soh, and Myoungkyu Song. SSDTutor: A feedback-driven intelligent tutoring system for secure software development. *Science of Computer Programming*, 227(102933), 2023. doi:10.1016/j.scico.2023.102933. (cited on page 260.)

[Och57] Akira Ochiai. Zoogeographical Studies on the Soleoid Fishes Found in Japan and its Neighhouring Regions-II. *Bulletin of the Japanese Society of Scientific Fisheries*, 22(9):526–530, 1957. doi:10.2331/suisan.22.526. (cited on page 208.)

[OGKY20] Shuyin OuYang, Fan Ge, Li Kuang, and Yuyu Yin. API Misuse Detection Based on Stacked LSTM. In *Proceedings of the 16th International Conference on Collaborative Computing: Networking, Applications and Worksharing (EAI)*, volume 349, pages 421–438. Springer, 2020. doi:10.1007/978-3-030-67537-0_26. (cited on pages 96, 97, 100, 101, and 267.)

[OHG+25] Lina Ochoa, Muhammad Hammad, Görkem Giray, Önder Babur, and Kwabena Bennin. Characterising harmful API uses and repair techniques: Insights from a systematic review. *Computer Science Review*, 57(100732), 2025. doi:10.1016/j.cosrev.2025.100732. (cited on pages 46, 47, and 48.)

[OLR+18] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. API Blindspots: Why Experienced Developers Write Vulnerable Code. In *Proceedings of the 14th Symposium on Usable Privacy and Security (SOUPS)*, pages 315–328. USENIX,

August 2018. URL: https://www.usenix.org/conference/soups2018/presentation/oliveira. (cited on pages 54, 102, and 250.)

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACM, 1984. doi:10.1145/800020.808263. (cited on page 28.)

[OS07] Frank Ortmeier and Gerhard Schellhorn. Formal Fault Tree Analysis - Practical Experiences. *Electronic Notes in Theoretical Computer Science*, 185:139–151, 2007. doi:10.1016/j.entcs.2007.05.034. (cited on page 4.)

[PBTL99a] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering Frequent Closed Itemsets for Association Rules. In *Proceedings of the 7th International Conference on Database Theory (ICDT)*, pages 398–416. Springer, 1999. doi:10.1007/3-540-49257-7_25. (cited on page 38.)

[PBTL99b] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1):25–46, 1999. doi:10.1016/S0306-4379(99)00003-4. (cited on page 38.)

[PCY95] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An Effective Hash-Based Algorithm for Mining Association Rules. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data (MOD)*, pages 175–186. ACM, 1995. doi:10.1145/223784.223813. (cited on page 38.)

[PDHR23] Nikhil Patnaik, Andrew Dwyer, Joseph Hallett, and Awais Rashid. SLR: From Saltzer and Schroeder to 2021. . . 47 Years of Research on the Development and Validation of Security API Recommendations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 32(3), April 2023. doi:10.1145/3561383. (cited on pages 4, 46, 47, 70, and 71.)

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-Directed Random Testing for Java. In *Proceedings of the 22nd Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 815–816. ACM, 2007. doi:10.1145/1297846.1297902. (cited on page 99.)

[PFM13] Marco Piccioni, Carlo A. Furia, and Bertrand Meyer. An Empirical Study of API Usability. In *Proceedings of the 7th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 5–14. IEEE, 2013. doi:10.1109/ESEM.2013.14. (cited on pages 54, 59, and 251.)

[PG09] Michael Pradel and Thomas R. Gross. Automatic Generation of Object Usage Specifications from Large Method Traces. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE)*, pages 371–382. IEEE, 2009. doi:10.1109/ASE.2009.60. (cited on pages 2, 5, and 99.)

[PG12] Michael Pradel and Thomas R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 288–298. IEEE, 2012. doi:10.1109/ICSE.2012.6227185. (cited on pages 5, 96, 99, 100, 101, and 267.)

[PHM00] Jian Pei, Jiawei Han, and Runying Mao. CLOSET: an efficient algorithm for mining frequent closed itemsets. In *Proceedings of the 5th Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD)*, pages 21–30. ACM, 2000. URL: https://cs.rhodes.edu/~welshc/COMP465_S15/Papers/closet-pei.pdf. (cited on page 38.)

[PHM+01] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 215–224. IEEE, 2001. doi:10.1109/ICDE.2001.914830. (cited on page 39.)

[PHR14] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, pages 212–222. ACM, 2014. doi:10.1145/2597008.2597152. (cited on pages 80 and 261.)

[PHR19] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries. In *Proceedings of the 15th Symposium on Usable Privacy and Security (SOUPS)*, pages 245–257. USENIX, August 2019. URL: https://www.usenix.org/conference/soups2019/presentation/patnaik. (cited on pages 4, 54, and 252.)

[PJAG12] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 925–935. IEEE, 2012. doi:10.1109/ICSE.2012.6227127. (cited on pages 96, 99, 101, and 267.)

[PMK+24] Justyna Petke, Matias Martinez, Maria Kechagia, Aldeida Aleti, and Federica Sarro. The Patch Overfitting Problem in Automated Program Repair: Practical Magnitude and a Baseline for Realistic Benchmarking. In *Proceedings of the 32nd Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 452–456. ACM, 2024. doi:10.1145/3663529.3663776. (cited on pages 208 and 246.)

[PP94] S. Paul and A. Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering (TSE)*, 20(6):463–475, 1994. doi:10.1109/32.295894. (cited on page 32.)

[PSM+07] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A Survey of Literature on the Teaching of Introductory Programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, pages 204–223. ACM, 2007. doi:10.1145/1345443.1345441. (cited on page 4.)

[QLAR15] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 24–36. ACM, 2015. doi:10.1145/2771783.2771791. (cited on pages 208 and 211.)

[QLL16] Dong Qiu, Bixin Li, and Hareton Leung. Understanding the API usage in Java. *Elsevier Journal of Information and Software Technology (IST)*, 73:81–100, 2016. doi:10.1016/j.infsof.2016.01.011. (cited on pages 4, 54, and 251.)

[QML+14] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 254–265. ACM, 2014. doi:10.1145/2568225.2568254. (cited on page 211.)

[QZW19] Xue Qin, Hao Zhong, and Xiaoyin Wang. TestMig: migrating GUI test cases from iOS to Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 284–295. ACM, 2019. doi:10.1145/3293882.3330575. (cited on page 35.)

[RAB+20] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, Breno Bernard Nicolau de França, Carlo Alberto Furia, Greg Gay, Nicolas Gold, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara Kitchenham, Valentina Lenarduzzi, Jorge Martínez, Jorge Melegati, Daniel Mendez, Tim Menzies, Jefferson Molleri, Dietmar Pfahl, Romain Robbes, Daniel Russo, Nyyti Saarimäki, Federica Sarro, Davide Taibi, Janet Siegmund, Diomidis Spinellis, Miroslaw Staron, Klaas Stol, Margaret-Anne Storey, Davide Taibi, Damian Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, Xiaofeng Wang, and Sira Vegas. Empirical standards for software engineering research, 2020. doi:10.48550/ARXIV.2010.03525. (cited on pages 16, 17, 19, 45, 63, 73, 91, 92, 143, 205, and 206.)

[RAT+06] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling. What do we know about defect detection methods? *IEEE Software*, 23(3):82–90, 2006. doi:10.1109/MS.2006.89. (cited on pages 5 and 25.)

[RBK+13] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API Property Inference Techniques. *IEEE*

Transactions on Software Engineering (TSE), 39(5):613–637, 2013. doi:10.1109/TSE.2012.63. (cited on pages 5, 15, 35, 36, 39, 70, 71, 93, 98, 103, 108, 140, and 199.)

[RC15] Martin P. Robillard and Yam B. Chhetri. Recommending reference API documentation. *Springer Empirical Software Engineering (EMSE)*, 20(6):1558–1586, 2015. doi:10.1007/S10664-014-9323-Y. (cited on pages 79 and 260.)

[RD11] Martin P Robillard and Robert DeLine. A field Study of API Learning Obstacles. *Springer Empirical Software Engineering (EMSE)*, 16(6):703–732, 2011. doi:10.1007/s10664-010-9150-8. (cited on pages 54 and 255.)

[RHR+25] Arjun Ramesh, Tianshu Huang, Jaspreet Riar, Ben L. Titzer, and Anthony Rowe. Unveiling Heisenbugs with Diversified Execution. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1), April 2025. doi:10.1145/3720428. (cited on page 139.)

[RKS+21] Mikko Raatikainen, Elina Kettunen, Ari Salonen, Marko Komssi, Tommi Mikkonen, and Timo Lehtonen. State of the Practice in Application Programming Interfaces (APIs): A Case Study. In *Proceedings of the 15th European Conference on Software Architecture (ECSA)*, volume 12857, pages 191–206. Springer, 2021. doi:10.1007/978-3-030-86044-8_14. (cited on page 54.)

[RLC20] Hao Ren, Yanhui Li, and Lin Chen. An Empirical Study on Critical Blocking Bugs. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC)*, pages 72–82. ACM, 2020. doi:10.1145/3387904.3389267. (cited on page 24.)

[RML+23] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. MELT: Mining Effective Lightweight Transformations from Pull Requests. In *Proceedings of the 38th International Conference on Automated Software Engineering (ASE)*, pages 1516–1528. IEEE, 2023. doi:10.1109/ASE56229.2023.00117. (cited on pages 35 and 199.)

[Rob09] Martin P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27–34, 2009. doi:10.1109/MS.2009.193. (cited on pages 4, 54, and 250.)

[RRL16] Mohammad Masudur Rahman, Chanchal K. Roy, and David Lo. RACK: Automatic API Recommendation Using Crowdsourced Knowledge. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 349–359. IEEE, 2016. doi:10.1109/SANER.2016.80. (cited on page 32.)

[RRS23] Ita Ryan, Utz Roedig, and Klaas-Jan Stol. Unhelpful Assumptions in Software Security Research. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, pages 3460–3474. ACM, 2023. doi:10.1145/3576915.3623122. (cited on pages 2, 70, and 71.)

[RRWF25] Joseph Renzullo, Pemma Reiter, Westley Weimer, and Stephanie Forrest. Automated Program Repair: Emerging Trends Pose and Expose Problems for Benchmarks. *ACM Computing Surveys*, February 2025. Just Accepted. doi:10.1145/3704997. (cited on pages 206, 209, 212, and 247.)

[RTP19] Irum Rauf, Elena Troubitsyna, and Ivan Porres. A systematic mapping study of API usability evaluation methods. *Computer Science Review*, 33:49–68, 2019. doi:10.1016/j.cosrev.2019.05.001. (cited on pages 4, 46, 47, 70, and 71.)

[RvV17] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning and impact of breaking changes in the Maven repository. *Elsevier Journal of Systems and Software (JSS)*, 129:140–158, 2017. doi:10.1016/j.jss.2016.04.008. (cited on pages 35, 79, and 261.)

[RXA+19] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, pages 2455–2472. ACM, 2019. doi:10.1145/3319535.3345659. (cited on page 100.)

[RYX+20] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *Proceedings of the 35th International Conference on Automated Software Engineering (ASE)*, pages 461–472. IEEE, 2020. doi:10.1145/3324884.3416551. (cited on pages 96, 98, 99, 101, and 267.)

[SA96] Ramakrishnan Srikant and Rakesh Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT)*, volume 1057, pages 3–17. Springer, 1996. doi:10.1007/BFb0014140. (cited on page 39.)

[SAB18] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 26th Joint Meeting of the European Software Engineering Conference/-Foundations of Software Engineering (ESEC/FSE)*, pages 908–911. ACM, 2018. doi:10.1145/3236024.3264598. (cited on page 34.)

[SAE+18] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from Building Static Analysis Tools at Google. *Communications of the ACM*, 61(4):58–66, March 2018. doi:10.1145/3188720. (cited on pages 5 and 108.)

[SAI20] Gian Luca Scoccia, Marco Autili, and Paola Inverardi. A self-configuring and adaptive privacy-aware permission system for Android apps. In *Proceedings of the 1st International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 38–47. IEEE, 2020. doi:10.1109/ACSOS49614.2020.00024. (cited on pages 79 and 260.)

[SAKW21] Jonathan Sharman, Claudia Acemyan, Philip Kortum, and Dan Wallach. Bad Tools Hurt: Lessons for teaching computer security skills to undergraduates. *International Journal of Computer Science Education in Schools*, 5(2):74–92, December 2021. doi:10.21585/ijcses.v5i2.131. (cited on pages 78, 79, 258, and 260.)

[SAvDB18] Anand Ashok Sawant, Maurício Aniche, Arie van Deursen, and Alberto Bacchelli. Understanding developers' needs on deprecation as a language feature. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 561–571. ACM, 2018. doi:10.1145/3180155.3180170. (cited on page 34.)

[SB22] Qusay Idrees Sarhan and Árpád Beszédes. A Survey of Challenges in Spectrum-Based Software Fault Localization. *IEEE Access*, 10:10618–10639, 2022. doi:10.1109/ACCESS.2022.3144079. (cited on page 208.)

[SBLGB15] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 532–543. ACM, 2015. doi:10.1145/2786805.2786825. (cited on page 208.)

[SBLV+20] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Valentina Piantadosi, Michele Lanza, and Rocco Oliveto. API Compatibility Issues in Android: Causes and Effectiveness of Data-driven Detection Techniques. *Springer Empirical Software Engineering (EMSE)*, 25(6):5006–5046, 2020. doi:10.1007/s10664-020-09877-w. (cited on page 158.)

[SC06] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: mining For Sample Code. In *Proceedings of the 21st Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 413–430. ACM, 2006. doi:10.1145/1167473.1167508. (cited on page 32.)

[SC07] Jeffrey Stylos and Steven Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 529–539. IEEE, 2007. doi:10.1109/ICSE.2007.92. (cited on pages 54 and 254.)

[Sch24] Samuel David Schwartz. *Empirical Quantitative Analyses of Research Software Engineering Projects in Scientific Computing*. PhD thesis, College of Arts and Sciences, University of Oregon, 6 2024. URL: https://www.cs.uoregon.edu/Reports/PHD-202406-Schwartz.pdf. (cited on pages 33 and 34.)

[SDSE20] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *Communications of the ACM*, 63(12):54–63, November 2020. doi:10.1145/3381831. (cited on page 247.)

[Sea07] Mícheál Ó Searcóid. *Metric Spaces*, chapter Metrics, pages 1–20. Springer, 2007. doi:10.1007/978-1-84628-627-8_1. (cited on page 152.)

[SED14] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. Solving the Search for Source Code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3), June 2014. doi:10.1145/2581377. (cited on page 32.)

[Ser19] Francesc Serratosa. Graph edit distance: Restrictions to be a metric. *Elsevier Pattern Recognition*, 90, 06 2019. doi:10.1016/j.patcog.2019.01.043. (cited on page 153.)

[SF83] Alberto Sanfeliu and King-Sun Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3):353–362, 1983. doi:10.1109/TSMC.1983.6313167. (cited on pages 148 and 153.)

[SFDB07] Zachary M. Saul, Vladimir Filkov, Premkumar Devanbu, and Christian Bird. Recommending Random Walks. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 15–24. ACM, 2007. doi:10.1145/1287624.1287629. (cited on page 32.)

[SFYM09] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A. Myers. Improving API Documentation Using API Usage Information. In *Proceedings of the 26th IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 119–126. IEEE, 2009. doi:10.1109/VLHCC.2009.5295283. (cited on pages 79, 259, and 260.)

[SGM20] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Modern Deep Learning Research. *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, 34(09):13693–13696, April 2020. doi:10.1609/aaai.v34i09.7123. (cited on page 247.)

[SHA14] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP)*, volume 8586, pages 157–181. Springer, 2014. doi:10.1007/978-3-662-44202-9_7. (cited on pages 78 and 259.)

[SJM08] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 471–480. ACM, 2008. doi:10.1145/1368088.1368153. (cited on pages 35 and 199.)

[SJR+15] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE, 2015. doi:10.1109/ASE.2015.86. (cited on page 26.)

[SJS+11] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. A General Software Defect-Proneness Prediction Framework. *IEEE Transactions on Software Engineering (TSE)*, 37(3):356–370, 2011. doi:10.1109/TSE.2010.90. (cited on page 25.)

[SK12] Thomas Scheller and Eva Eva Kühn. Influencing Factors on the Usability of API Classes and Methods. In *Proceedings of the 19th International Conference on Engineering of Computer-Based Systems (ECBS)*, pages 232–241. IEEE, 2012. doi:10.1109/ECBS.2012.27. (cited on page 54.)

[SK13] Thomas Scheller and Eva Kühn. Usability Evaluation of Configuration-Based API Design Concepts. In *Proceedings of the 1st International Conference on Human Factors in Computing and Informatics (SouthCHI)*, volume 7946, pages 54–73. Springer, 2013. doi:10.1007/978-3-642-39062-3_4. (cited on page 54.)

[SK15] Thomas Scheller and Eva Kühn. Automated measurement of API usability: The API Concepts Framework. *Elsevier Journal of Information and Software Technology (IST)*, 61:145–162, 2015. doi:10.1016/j.infsof.2015.01.009. (cited on pages 54 and 254.)

[SKG21] Christoph Schröer, Felix Kruse, and Jorge Marx Gómez. A Systematic Literature Review on Applying CRISP-DM Process Model. *Procedia Computer Science*, 181:526–534, 2021. CENTERIS 2020 - International Conference on ENTERprise Information Systems / ProjMAN 2020 - International Conference on Project MANagement / HCist 2020 - International Conference on Health and Social Care Information Systems and Technologies 2020, CENTERIS/ProjMAN/HCist 2020. doi:10.1016/j.procs.2021.01.199. (cited on pages 41 and 42.)

[SKP14] Ripon K. Saha, Sarfraz Khurshid, and Dewayne E. Perry. An Empirical Study of Long Lived Bugs. In *Proceedings of the 1st Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 144–153. IEEE, 2014. doi:10.1109/CSMR-WCRE.2014.6747164. (cited on page 24.)

[SLL+18] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. Bugs.jar: A Large-scale, Diverse Dataset of Real-world Java Bugs. In *Proceedings of the 15th International Working Conference on Mining Software Repositories (MSR)*, pages 10–13. ACM, 2018. doi:10.1145/3196398.3196473. (cited on pages 209 and 220.)

[SLP+09] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. ASSURE: Automatic Software Self-healing Using REscue points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48. ACM, 2009. doi:10.1145/1508244.1508250. (cited on page 210.)

[SM08] Jeffrey Stylos and Brad A. Myers. The Implications of Method Placement on API Learnability. In *Proceedings of the 16th Joint Meeting of*

the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE), pages 105–112. ACM, 2008. doi:10.1145/1453101.1453117. (cited on pages 4, 54, 78, 253, 254, and 259.)

[SM17] André L. Santos and Brad A. Myers. Design annotations to improve API discoverability. *Elsevier Journal of Systems and Software (JSS)*, 126:17–33, 2017. doi:10.1016/j.jss.2016.12.036. (cited on pages 79, 259, and 260.)

[SMAR17] S M Sohan, Frank Maurer, Craig Anslow, and Martin P. Robillard. A Study of the Effectiveness of Usage Examples in REST API Documentation. In *Proceedings of the 34th IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 53–61. IEEE, 2017. doi:10.1109/VLHCC.2017.8103450. (cited on pages 4, 54, and 252.)

[Som18] Ian Sommerville. *Software Engineering*, volume 10. Pearson Deutschland GmbH, 2018. ISBN-13: 978-3-86894-344-3. (cited on pages 1, 4, 14, 25, 26, 27, and 33.)

[SPK14] Anirudh Santhiar, Omesh Pandita, and Aditya Kanade. Mining Unit Tests for Discovery and Migration of Math APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(1), October 2014. doi:10.1145/2629506. (cited on pages 78, 79, 260, and 261.)

[SRB19] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. To react, or not to react: Patterns of reaction to API deprecation. *Springer Empirical Software Engineering (EMSE)*, 24(6):3824–3870, 2019. doi:10.1007/S10664-019-09713-W. (cited on page 35.)

[SSA15] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on Internal and External Validity in Empirical Software Engineering. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, volume 1, pages 9–19. IEEE, May 2015. doi:10.1109/ICSE.2015.24. (cited on pages 92 and 233.)

[SSC+18] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th International Conference on Software Engineering - Software Engineering in Practice (ICSE-SEIP)*, pages 181–190. ACM, 2018. doi:10.1145/3183519.3183525. (cited on page 4.)

[SSD15] Mohamed Aymen Saied, Houari Sahraoui, and Bruno Dufour. An observational study on API usage constraints and their documentation. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 33–42. IEEE, 2015. doi:10.1109/SANER.2015.7081813. (cited on pages 54 and 253.)

[SSE15] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How Developers Search for Code: A Case Study. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 191–201. ACM, 2015. doi:10.1145/2786805.2786855. (cited on page 32.)

[SSNB22] Michael Schlichtig, Steffen Sassalla, Krishna Narasimhan, and Eric Bodden. FUM - A Framework for API Usage constraint and Misuse Classification. In *Proceedings of the 29th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 673–684. IEEE, 2022. doi:10.1109/SANER53432.2022.00085. (cited on pages 2 and 26.)

[STFR17] Michael Stauffer, Thomas Tschachtli, Andreas Fischer, and Kaspar Riesen. A Survey on Applications of Bipartite Graph Edit Distance. In *Proceedings of the 10th International Workshop on Graph-Based Representations in Pattern Recognition*, volume 10310. Springer, 2017. doi:10.1007/978-3-319-58961-9_22. (cited on page 153.)

[SWH14] Wei Shen, Jianyong Wang, and Jiawei Han. *Sequential Pattern Mining*, pages 261–282. Springer, Cham, 2014. doi:10.1007/978-3-319-07821-2_11. (cited on pages 36 and 39.)

[SWT+21] Fabio Santos, Igor Wiese, Bianca Trinkenreich, Igor Steinmacher, Anita Sarma, and Marco A. Gerosa. Can I Solve It? Identifying APIs Required to Complete OSS Tasks. In *Proceedings of the 18th International Working Conference on Mining Software Repositories (MSR)*, pages 346–357. IEEE, 2021. doi:10.1109/MSR52588.2021.00047. (cited on page 258.)

[SWZ+20] Qi Shen, Shijun Wu, Yanzhen Zou, Zixiao Zhu, and Bing Xie. From API to NLI: A new interface for library reuse. *Elsevier Journal of Systems and Software (JSS)*, 169(110728), 2020. doi:10.1016/j.jss.2020.110728. (cited on pages 78 and 259.)

[SXC+19] Jiamou Sun, Zhenchang Xing, Rui Chu, Heilai Bai, Jinshui Wang, and Xin Peng. Know-How in Programming Tasks: From Textual Tutorials to Task-Oriented Knowledge Graph. In *Proceedings of the 35th International Conference on Software Maintenance and Evolution (ICSME)*, pages 257–268. IEEE, 2019. doi:10.1109/ICSME.2019.00039. (cited on page 259.)

[SXP+21] Jiamou Sun, Zhenchang Xing, Xin Peng, Xiwei Xu, and Liming Zhu. Task-Oriented API Usage Examples Prompting Powered By Programming Task Knowledge Graph. In *Proceedings of the 37th International Conference on Software Maintenance and Evolution (ICSME)*, pages 448–459. IEEE, 2021. doi:10.1109/ICSME52107.2021.00046. (cited on pages 79 and 260.)

[SY21] Mahsa Hasani Sadi and Eric Yu. RAPID: a knowledge-based assistant for designing web apis. *Requirements Engineering*, 26(2):185–236, 2021. doi:10.1007/S00766-020-00342-0. (cited on pages 78 and 258.)

[Szp30] Edward Szpilrajn. Sur l'extension de l'ordre partiel. *Fundamenta Mathematicae*, 16(1):386–389, 1930. URL: http://eudml.org/doc/212499. (cited on page 233.)

[SZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When Do Changes Induce Fixes? In *Proceedings of the 2nd International Working Conference on Mining Software Repositories (MSR)*, pages 1–5. ACM, 2005. doi:10.1145/1082983.1083147. (cited on pages 33, 116, and 229.)

[TCG+09] Marisa Thoma, Hong Cheng, Arthur Gretton, Jiawei Han, Hans-Peter Kriegel, Alex Smola, Le Song, Philip S. Yu, Xifeng Yan, and Karsten Borgwardt. Near-optimal supervised feature selection among frequent subgraphs. In *Proceedings of the 9th SIAM International Conference on Data Mining (SDM)*, pages 1076–1087. SIAM, 2009. doi:10.1137/1.9781611972795.92. (cited on page 104.)

[TCK21] Kyle Thayer, Sarah E. Chasins, and Amy J. Ko. A Theory of Robust API Knowledge. *ACM Transactions on Computing Education (TOCE)*, 21(1), January 2021. doi:10.1145/3444945. (cited on pages 4, 54, 251, and 254.)

[TFB13] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. Automatic discovery of function mappings between similar libraries. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 192–201. IEEE, 2013. doi:10.1109/WCRE.2013.6671294. (cited on pages 35 and 199.)

[THLH21] Christopher Steven Timperley, Lauren Herckis, Claire Le Goues, and Michael Hilton. Understanding and improving artifact sharing in software engineering research. *Springer Empirical Software Engineering (EMSE)*, 26(6), 2021. doi:10.1007/S10664-021-09973-5. (cited on page 245.)

[TLL13] Ferdian Thung, David Lo, and Julia Lawall. Automated Library Recommendation. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 182–191. IEEE, 2013. doi:10.1109/WCRE.2013.6671293. (cited on page 32.)

[TM10] Nikolaj Tatti and Michael Mampaey. Using background knowledge to rank itemsets. *Springer Data Mining and Knowledge Discovery*, 21(2):293–309, 2010. doi:10.1007/s10618-010-0188-4. (cited on page 41.)

[TMC14] Nikolaj Tatti, Fabian Moerchen, and Toon Calders. Finding robust itemsets under subsampling. *ACM Transactions on Database Systems (TODS)*, 39(3):20:1–20:27, 2014. doi:10.1145/2656261. (cited on page 40.)

[TR16] Christoph Treude and Martin P. Robillard. Augmenting API Documentation with Insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 392–403. ACM, 2016. doi:10.1145/2884781.2884800. (cited on pages 79 and 260.)

[TRJ+21] Yixuan Tang, Zhilei Ren, He Jiang, Xiao-Chen Li, and Weiqiang Kong. An Empirical Comparison Between Tutorials and Crowd Documentation of Application Programming Interface. *Journal of Computer Science and Technology*, 36(4):856–876, 2021. doi:10.1007/S11390-020-0042-0. (cited on pages 79 and 260.)

[TS21] Valerio Terragni and Pasquale Salza. APIzation: Generating Reusable APIs from StackOverflow Code Snippets. In *Proceedings of the 36th International Conference on Automated Software Engineering (ASE)*, pages

542–554. IEEE, 2021. doi:10.1109/ASE51524.2021.9678576. (cited on pages 78 and 259.)

[TSM18] Swapna Thorve, Chandani Sreshtha, and Na Meng. An Empirical Study of Flaky Tests in Android Apps. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME)*, pages 534–538. IEEE, 2018. doi:10.1109/ICSME.2018.00062. (cited on page 24.)

[TVBW21] Mohammad Tahaei, Kami Vaniea, Konstantin (Kosta) Beznosov, and Maria K Wolters. Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them. In *Proceedings of the 39th Conference on Human Factors in Computing Systems (CHI)*, pages 1–17. ACM, 2021. doi:10.1145/3411764.3445616. (cited on pages 78 and 258.)

[TWB+19] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4), September 2019. doi:10.1145/3340544. (cited on page 212.)

[TWLL13] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. Automatic Recommendation of API Methods from Feature Requests. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE)*, pages 290–300. IEEE, 2013. doi:10.1109/ASE.2013.6693088. (cited on page 32.)

[TX09a] Suresh Thummalapenta and Tao Xie. Alattin: Mining Alternative Patterns for Detecting Neglected Conditions. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE)*, pages 283–294. IEEE, 2009. doi:10.1109/ASE.2009.72. (cited on pages 5, 96, 99, 100, 101, 109, and 267.)

[TX09b] Suresh Thummalapenta and Tao Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 496–506. IEEE, 2009. doi:10.1109/ICSE.2009.5070548. (cited on pages 95, 100, 101, 109, and 267.)

[TZKV18] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding Sparkle to Social Coding: An Empirical Study of Repository Badges in the npm Ecosystem. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 511–522. ACM, 2018. doi:10.1145/3180155.3180209. (cited on pages 78, 258, and 259.)

[UBv+22] Martin Ukrop, Michaela Balážová, Pavol Žáčik, Eric Vincent Valčík, and Vashek Matyas. Assessing Real-World Applicability of Redesigned Developer Documentation for Certificate Validation Errors. In *Proceedings of the 7th European Symposium on Usable Security (EuroUSEC)*, pages 131–144. ACM, 2022. doi:10.1145/3549015.3554296. (cited on pages 79 and 260.)

[UKR20] Gias Uddin, Foutse Khomh, and Chanchal K Roy. Mining API usage scenarios from stack overflow. *Elsevier Journal of Information and Software Technology (IST)*, 122(106277), 2020. doi:10.1016/j.infsof.2020.106277. (cited on pages 79 and 260.)

[UKR21] Gias Uddin, Foutse Khomh, and Chanchal K. Roy. Automatic API Usage Scenario Documentation from Technical Q&A Sites. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3), April 2021. doi:10.1145/3439769. (cited on pages 79 and 260.)

[UR15] Gias Uddin and Martin P. Robillard. How API Documentation Fails. *IEEE Software*, 32(4):68–75, 2015. doi:10.1109/MS.2014.80. (cited on page 54.)

[VFM15] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. An Empirical Study of Bugs in Test Code. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110. IEEE, 2015. doi:10.1109/ICSM.2015.7332456. (cited on page 24.)

[Vid22] Melina Vidoni. A systematic process for Mining Software Repositories: Results from a systematic literature review. *Elsevier Journal of Information and Software Technology (IST)*, 144(106791), 2022. doi:10.1016/j.infsof.2021.106791. (cited on pages 33 and 34.)

[VKC21] Aparna Vadlamani, Rishitha Kalicheti, and Sridhar Chimalakonda. APIScanner - Towards Automated Detection of Deprecated APIs in Python Libraries. In *Proceedings of the 43rd Companion of the International Conference on Software Engineering (ICSEC)*, pages 5–8. IEEE/ACM, 2021. doi:10.1109/ICSE-Companion52605.2021.00022. (cited on pages 35 and 198.)

[VL16a] Sebastián Ventura and José María Luna. *Introduction to Pattern Mining*, pages 1–26. Springer, Cham, 2016. doi:10.1007/978-3-319-33858-3_1. (cited on page 36.)

[VL16b] Sebastián Ventura and José María Luna. *Pattern Mining with Genetic Algorithms*, pages 63–85. Springer, Cham, 2016. doi:10.1007/978-3-319-33858-3_4. (cited on page 37.)

[VT14] Jilles Vreeken and Nikolaj Tatti. *Interesting Patterns*, pages 105–134. Springer, Cham, 2014. doi:10.1007/978-3-319-07821-2_5. (cited on pages 36, 37, and 40.)

[WA19] Chamila Wijayarathna and Nalin Asanka Gamagedara Arachchilage. Why Johnny can't develop a secure application? A usability analysis of Java Secure Socket Extension API. *Computer & Security (CS)*, 80:54–73, 2019. doi:10.1016/j.cose.2018.09.007. (cited on pages 54 and 254.)

[Wag13] Stefan Wagner. *Software Product Quality Control*. Springer, 2013. ISBN-13: 978-3-642-38570-4. doi:10.1007/978-3-642-38571-1. (cited on page 25.)

[WBI+23] Wengran Wang, John Bacher, Amy Isvik, Ally Limke, Sandeep Sthapit, Yang Shi, Benyamin T. Tabarsi, Keith Tran, Veronica Cateté, Tiffany Barnes, Chris Martens, and Thomas Price. Investigating the Impact of On-Demand Code Examples on Novices' Open-Ended Programming Experience. In *Proceedings of the 19th International Computing Education Research Conference (ICER)*, pages 464–475. ACM, 2023. doi:10.1145/3568813.3600141. (cited on pages 79 and 260.)

[WBJS20] Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. An Empirical Study on Regular Expression Bugs. In *Proceedings of the 17th International Working Conference on Mining Software Repositories (MSR)*, pages 103–113. ACM, 2020. doi:10.1145/3379597.3387464. (cited on page 24.)

[WBK21] Pei Wang, Julian Bangert, and Christoph Kern. If It's Not Secure, It Should Not Compile: Preventing DOM-Based XSS in Large-Scale Web Development with API Hardening. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 1360–1372. IEEE, 2021. doi:10.1109/ICSE43902.2021.00123. (cited on pages 78, 258, and 259.)

[WCH+20] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *Proceedings of the 36th International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020. doi:10.1109/ICSME46990.2020.00014. (cited on pages 1 and 14.)

[Wei06] Westley Weimer. Patches as Better Bug Reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 181–190. ACM, 2006. doi:10.1145/1173706.1173734. (cited on pages 213, 230, and 231.)

[WG24] David Gray Widder and Claire Le Goues. What Is a 'Bug'? *Communications of the ACM*, 67(11):32–34, October 2024. doi:10.1145/3662730. (cited on page 23.)

[WGAK10] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. AURA: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 325–334. ACM, 2010. doi:10.1145/1806799.1806848. (cited on pages 35 and 199.)

[WGL+16] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering (TSE)*, 42(8):707–740, 2016. doi:10.1109/TSE.2016.2521368. (cited on pages 4 and 25.)

[WGMC15] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection. In *Proceedings of the 24th ACM SIGSOFT International Sym-*

posium on Software Testing and Analysis (ISSTA), pages 71–82. ACM, 2015. doi:10.1145/2771783.2771795. (cited on pages 1 and 14.)

[WH04] Jianyong Wang and Jiawei Han. BIDE: Efficient Mining of Frequent Closed Sequences. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, pages 79–90. IEEE, 2004. doi:10.1109/ICDE.2004.1319986. (cited on page 39.)

[WHH+24] Moshi Wei, Nima Shiri Harzevili, Yuekai Huang, Jinqiu Yang, Junjie Wang, and Song Wang. Demystifying and Detecting Misuses of Deep Learning APIs. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, pages 1–12. ACM, 2024. doi:10.1145/3597503.3639177. (cited on pages 4 and 247.)

[WHW+22] Moshi Wei, Yuchao Huang, Junjie Wang, Jiho Shin, Nima Shiri Harzevili, and Song Wang. API Recommendation for Machine Learning Libraries: How Far Are We? In *Proceedings of the 30th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 370–381. ACM, 2022. doi:10.1145/3540250.3549124. (cited on page 32.)

[WJZ+21] Di Wu, Xiao-Yuan Jing, Hongyu Zhang, Bing Li, Yu Xie, and Baowen Xu. Generating API tags for tutorial fragments from Stack Overflow. *Springer Empirical Software Engineering (EMSE)*, 26(66), 2021. doi:10.1007/S10664-021-09962-8. (cited on pages 79 and 260.)

[WJZ+23] Di Wu, Xiao-Yuan Jing, Hongyu Zhang, Yang Feng, Haowen Chen, Yuming Zhou, and Baowen Xu. Retrieving API Knowledge from Tutorials and Stack Overflow Based on Natural Language Queries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 32(5), July 2023. doi:10.1145/3565799. (cited on pages 78, 79, 258, and 260.)

[WKA+16] Wei Wu, Foutse Khomh, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of api changes and usages based on apache and eclipse ecosystems. *Springer Empirical Software Engineering (EMSE)*, 21(6):2366–2412, 2016. doi:10.1007/s10664-015-9411-7. (cited on page 54.)

[WLC16] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 226–237. IEEE/ACM, 2016. doi:10.1145/2970276.2970312. (cited on page 158.)

[WLC19] L. Wei, Y. Liu, and S. Cheung. PIVOT: Learning API-device correlations to facilitate android compatibility issue detection. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 878–888. IEEE/ACM, 2019. doi:10.1109/ICSE.2019.00094. (cited on page 158.)

[WLC+20] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Liu Xuanzhe. Understanding and Detecting Fragmentation-Induced Compatibility Issues for Android Apps. *IEEE Transactions on Software Engineering (TSE)*, 46(11):1176–1199, 2020. doi:10.1109/TSE.2018.2876439. (cited on page 158.)

[WLLC20] Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. Exploring how deprecated Python library APIs are (not) handled. In *Proceedings of the 28th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 233–244. ACM, 2020. doi:10.1145/3368089.3409735. (cited on pages 35, 54, and 252.)

[WLY+24] Ruixin Wang, Minghai Lu, Cody Hao Yu, Yi-Hsiang Lai, and Tianyi Zhang. Automated Deep Learning Optimization via DSL-Based Source Code Transformation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 479–490. ACM, 2024. doi:10.1145/3650212.3652143. (cited on pages 79 and 260.)

[WN05] Westley Weimer and George C. Necula. Mining Temporal Specifications for Error Detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440, pages 461–476. Springer, 2005. doi:10.1007/978-3-540-31980-1_30. (cited on pages 2, 5, and 36.)

[Woh14] Claes Wohlin. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–10. ACM, 2014. doi:10.1145/2601248.2601268. (cited on pages 7, 45, 48, 49, 50, 71, and 88.)

[WPWZ19] Shaohua Wang, NhatHai Phan, Yan Wang, and Yong Zhao. Extracting API Tips from Developer Question and Answer Websites. In *Proceedings of the 16th International Working Conference on Mining Software Repositories (MSR)*, pages 321–332. IEEE, 2019. doi:10.1109/MSR.2019.00058. (cited on pages 78, 79, 259, and 260.)

[WTH+22] Stefan Winter, Christopher S. Timperley, Ben Hermann, Jürgen Cito, Jonathan Bell, Michael Hilton, and Dirk Beyer. A Retrospective Study of One Decade of Artifact Evaluations. In *Proceedings of the 30th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 145–156. ACM, 2022. doi:10.1145/3540250.3549172. (cited on page 245.)

[WTH+24] Hironori Washizaki, Yatheendranath TJ, Rich Hilliard, Kenneth Nidiffer, Pete Brink, V.S. Mani, Hari Prasad Devarapalli, Annette Reilly, Narendra S Chowdhury, Dharanipragada Janakiram, Juan Garbajosa, Maria Isabel Sánchez Segura, Peter Leather, Andy Chen, and Steve Schwarm. *Guide to the Software Engineering Body of Knowledge V4.0*, volume 4. IEEE, 2024. URL: http://www.swebok.org. (cited on pages 1, 3, 4, 5, 14, and 25.)

[WXL+21] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. Smart Contract Security: A Practitioners' Perspective. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 1410–1422. IEEE, 2021. doi:10.1109/ICSE43902.2021.00127. (cited on pages 80 and 261.)

[WXQ23] Yulin Wu, Zhiwu Xu, and Shengchao Qin. Detecting API-Misuse Based on Pattern Mining via API Usage Graph with Parameters. In *Proceedings of the 17th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, volume 13931, pages 344–363. Springer, 2023. doi:10.1007/978-3-031-35257-7_21. (cited on pages 5, 95, 97, 99, 101, and 267.)

[WY22] Kai Wang and Ping Yu. AUGraft: Graft New API Usage into Old Code. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, pages 55–64. ACM, 2022. doi:10.1145/3545258.3545279. (cited on pages 35 and 199.)

[WZ11] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. *Springer Automated Software Engineering*, 18(3-4):263–292, 2011. doi:10.1007/S10515-011-0084-1. (cited on pages 5, 93, 95, 99, 101, 102, and 267.)

[WZ23] Xiaoke Wang and Lei Zhao. APICAD: Augmenting API Misuse Detection through Specifications from Code and Documents. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pages 245–256. IEEE, 2023. doi:10.1109/ICSE48619.2023.00032. (cited on pages 2, 5, 96, 98, 99, 101, 103, 109, and 267.)

[WZF+12] Lijie Wang, Yanzhen Zou, Lu Fang, Bing Xie, and Fuqing Yang. An Exploratory Study of API Usage Examples on the Web. In *Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 396–405. IEEE, 2012. doi:10.1109/APSEC.2012.122. (cited on pages 79 and 260.)

[WZL07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 35–44. ACM, 2007. doi:10.1145/1287624.1287632. (cited on pages 2, 5, 93, 95, 99, 101, 102, 105, 198, 200, and 267.)

[XBL+17] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. What do developers search for on the web? *Springer Empirical Software Engineering (EMSE)*, 22(6):3149–3185, 2017. doi:10.1007/S10664-017-9514-4. (cited on page 32.)

[XDM19] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: Inference and Application of API Migration Edits. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC)*, pages 335–346. IEEE, 2019. doi:10.1109/ICPC.2019.00052. (cited on pages 35 and 199.)

[XGW24] Qingxin Xu, Yu Gao, and Jun Wei. An Empirical Study on Kubernetes Operator Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, page 1746–1758. ACM, 2024. doi:10.1145/3650212.3680396. (cited on page 24.)

[XMD+17] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering (TSE)*, 43(1):34–55, 2017. doi:10.1109/TSE.2016.2560811. (cited on page 211.)

[XR17] Qi Xin and Steven P. Reiss. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 226–236. ACM, 2017. doi:10.1145/3092703.3092718. (cited on page 208.)

[XS06] Zhenchang Xing and Eleni Stroulia. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *Proceedings of the 22nd International Conference on Software Maintenance (ICSM)*, pages 458–468. IEEE, 2006. doi:10.1109/ICSM.2006.52. (cited on page 35.)

[XXS+23] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1319–1331. ACM, 2023. doi:10.1145/3597926.3598138. (cited on page 24.)

[Yat34] Frank Yates. Contingency Tables Involving Small Numbers and the chi2 Test. *Supplement to the Journal of the Royal Statistical Society*, 1(2):217–235, 1934. doi:10.2307/2983604. (cited on page 136.)

[YB20] Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering (TSE)*, 46(10):1040–1067, 2020. doi:10.1109/TSE.2018.2874648. (cited on page 211.)

[YEB+06] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 282–291. ACM, 2006. doi:10.1145/1134285.1134325. (cited on page 109.)

[YH02] Xifeng Yan and Jiawei Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proceedings of the 2nd International Conference on Data Mining (ICDM)*, pages 721–724. IEEE, 2002. doi:10.1109/ICDM.2002.1184038. (cited on pages 40 and 104.)

[YH03] Xifeng Yan and Jiawei Han. CloseGraph: Mining Closed Frequent Graph Patterns. In *Proceedings of the 9th International Conference on Knowledge*

Discovery and Data Mining (KDD), pages 286–295. ACM, 2003. doi:10.1145/956750.956784. (cited on page 40.)

[YHA03] Xifeng Yan, Jiawei Han, and Ramin Afshar. CloSpan: Mining Closed Sequential Patterns in Large Datasets. In *Proceedings of the 3rd SIAM International Conference on Data Mining (SDM)*, pages 166–177. SIAM, 2003. doi:10.1137/1.9781611972733.15. (cited on page 39.)

[YHWH24] Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. Ivie: Lightweight Anchored Explanations of Just-Generated Code. In *Proceedings of the 42nd Conference on Human Factors in Computing Systems (CHI)*, pages 1–15. ACM, 2024. doi:10.1145/3613904.3642239. (cited on pages 78, 247, and 259.)

[YHXF22] Yilin Yang, Tianxing He, Zhilong Xia, and Yang Feng. A comprehensive empirical study on bug characteristics of deep learning frameworks. *Elsevier Journal of Information and Software Technology (IST)*, 151(107004), 2022. doi:10.1016/j.infsof.2022.107004. (cited on pages 24 and 54.)

[YMS+16] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *Proceedings of the 25th USENIX Security Symposium*, pages 363–378. USENIX, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun. (cited on pages 95, 103, 109, and 267.)

[YRW22] Jingbo Yang, Jian Ren, and Wenjun Wu. API Misuse Detection Method Based on Transformer. In *Proceedings of the 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 958–969. IEEE, 2022. doi:10.1109/QRS57517.2022.00100. (cited on pages 2, 96, 100, 101, and 267.)

[ZBW+22] Zhou Zhou, Lili Bo, Xiaoxue Wu, Xiaobing Sun, Tao Zhang, Bin Li, Jiale Zhang, and Sicong Cao. SPVF: security property assisted vulnerability fixing via attention-based models. *Springer Empirical Software Engineering (EMSE)*, 27(171), 2022. doi:10.1007/S10664-022-10216-4. (cited on page 212.)

[ZCC+18] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 129–140. ACM, 2018. doi:10.1145/3213846.3213866. (cited on pages 24 and 54.)

[ZCSZ21] Hushuang Zeng, Jingxin Chen, Beijun Shen, and Hao Zhong. Mining API Constraints from Library and Client to Detect API Misuses. In *Proceedings of the 28th Asia-Pacific Software Engineering Conference (APSEC)*, pages 161–170. IEEE, 2021. doi:10.1109/APSEC53868.2021.00024. (cited on pages 5, 95, 98, 101, 104, 109, 198, 200, and 267.)

[ZDAT22] Oleksandr Zaitsev, Stéphane Ducasse, Nicolas Anquetil, and Arnaud Thiefaine. DepMiner: Automatic Recommendation of Transformation Rules for Method Deprecation. In *Reuse and Software Quality*, pages 22–37. Springer, 2022. doi:10.1007/978-3-031-08129-3_2. (cited on page 35.)

[Zel06] Andreas Zeller. CHAPTER 1 - How Failures Come to Be. In Andreas Zeller, editor, *Why Programs Fail*, pages 1–26. Morgan Kaufmann, Boston, first edition edition, 2006. doi:10.1016/B978-0-12-374515-6.00001-0. (cited on pages 3, 4, 14, 25, and 26.)

[ZER11] Minhaz F. Zibran, Farjana Z. Eishita, and Chanchal K. Roy. Useful, But Usable? Factors Affecting the Usability of APIs. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 151–155. IEEE, 2011. doi:10.1109/WCRE.2011.26. (cited on pages 4, 54, and 253.)

[ZFM+23] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A Survey of Learning-based Automated Program Repair. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 33(2), December 2023. doi:10.1145/3631974. (cited on pages 205, 206, 207, 208, 209, 211, 212, 232, 236, 245, and 247.)

[ZHKG20] Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L. Glassman. Enabling Data-Driven API Design with Community Usage Data: A Need-Finding Study. In *Proceedings of the 38th Conference on Human Factors in Computing Systems (CHI)*, pages 1–13. ACM, 2020. doi:10.1145/3313831.3376382. (cited on pages 4, 54, 250, and 253.)

[Zib08] Minhaz Fahim Zibran. What Makes APIs Difficult to Use? *International Journal of Computer Science and Network Security (IJCSNS)*, 8(4):255–261, 2008. URL: http://paper.ijcsns.org/07_book/200804/20080436.pdf. (cited on pages 46, 47, 70, and 71.)

[ZJR+21] Jingxuan Zhang, He Jiang, Zhilei Ren, Tao Zhang, and Zhiqiu Huang. Enriching API Documentation with Code Samples and Usage Scenarios from Crowd Knowledge. *IEEE Transactions on Software Engineering (TSE)*, 47(6):1299–1314, 2021. doi:10.1109/TSE.2019.2919304. (cited on pages 79 and 260.)

[ZM19] Hao Zhong and Hong Mei. An Empirical Study on API Usages. *IEEE Transactions on Software Engineering (TSE)*, 45(4):319–334, 2019. doi:10.1109/TSE.2017.2782280. (cited on pages 1 and 246.)

[ZS15] Hao Zhong and Zhendong Su. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 913–923. IEEE, 2015. doi:10.1109/ICSE.2015.101. (cited on pages 2, 24, and 27.)

[ZTW+09] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing Stars: On Approximating Graph Edit Distance.

Proceedings of the VLDB Endowment (PVLDB), 2(1):25–36, 2009. doi:10.14778/1687627.1687631. (cited on pages 148, 154, and 202.)

[ZTX+10] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 195–204. ACM, 2010. doi:10.1145/1806799.1806831. (cited on page 35.)

[ZUR+18] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. Are Code Examples on an Online Q&A Forum Reliable?: A Study of API Misuse on Stack Overflow. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 886–896. ACM, 2018. doi:10.1145/3180155.3180260. (cited on pages 1, 2, 54, 78, 102, 213, 252, 253, and 259.)

[ZW16] Jing Zhou and Robert J. Walker. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 24th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 266–277. ACM, 2016. doi:10.1145/2950290.2950298. (cited on pages 35 and 198.)

[ZWCX22] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys*, 54(11s), September 2022. doi:10.1145/3512345. (cited on page 26.)

[ZWH19] Alexander Zeier, Alexander Wiesmaier, and Andreas Heinemann. API Usability of Stateful Signature Schemes. In *Advances in Information and Computer Security*, pages 221–240. Springer, 2019. doi:10.1007/978-3-030-26834-3_13. (cited on pages 77 and 258.)

[ZWL+23] Pengzhan Zhao, Xiongfei Wu, Junjie Luo, Zhuo Li, and Jianjun Zhao. An Empirical Study of Bugs in Quantum Machine Learning Frameworks. In *Proceedings of the 2nd IEEE International Conference on Quantum Software (QSW)*, pages 68–75. IEEE, 2023. doi:10.1109/QSW59989.2023.00018. (cited on pages 24 and 54.)

[ZWX+23] Bingzhe Zhou, Xinying Wang, Shengbin Xu, Yuan Yao, Minxue Pan, Feng Xu, and Xiaoxing Ma. Hybrid API Migration: A Marriage of Small API Mapping Models and Large Language Models. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, pages 12–21. ACM, 2023. doi:10.1145/3609437.3609466. (cited on pages 35, 199, and 247.)

[ZWY+20] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald Gall. Automatic Detection and Repair Recommendation of Directive Defects in Java API Documentation. *IEEE Transactions on Software Engineering (TSE)*, 46(9):1004–1023, 2020. doi:10.1109/TSE.2018.2872971. (cited on pages 79 and 261.)

[ZXL+24] Dawen Zhang, Boming Xia, Yue Liu, Xiwei Xu, Thong Hoang, Zhenchang Xing, Mark Staples, Qinghua Lu, and Liming Zhu. Privacy and

Copyright Protection in Generative AI: A Lifecycle Perspective. In *Proceedings of the 3rd International Conference on AI Engineering - Software Engineering for AI (CAIN)*, pages 92—97. ACM, 2024. doi:10.1145/3644815.3644952. (cited on page 247.)

[ZXZ+09] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 318–343. Springer, 2009. doi:10.1007/978-3-642-03013-0_15. (cited on pages 5, 109, 112, and 136.)

[ZYLK19] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. Analyzing and Supporting Adaptation of Online Code Examples. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 316–327. IEEE, 2019. doi:10.1109/ICSE.2019.00046. (cited on pages 78, 79, 259, and 260.)

[ZZW+23] Chenguang Zhu, Mengshi Zhang, Xiuheng Wu, Xiufeng Xu, and Yi Li. Client-Specific Upgrade Compatibility Checking via Knowledge-Guided Discovery. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 32(4), May 2023. doi:10.1145/3582569. (cited on pages 79 and 261.)

[ZZXM09] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring Resource Specifications from Natural Language API Documentation. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE)*, pages 307–318. IEEE, 2009. doi:10.1109/ASE.2009.94. (cited on pages 93, 96, 99, 101, and 267.)

Erklärung über verwendete Hilfsmittel

Für die Erstellung dieser Dissertation wurden folgende, technische Hilfsmittel basierend auf sogenannter künstlicher Intelligenz (KI) verwendet:

Grammarly Pro (https://www.grammarly.com/pro) der Firma Grammarly, Inc.: Grammarly Pro wurde für die Überprüfung der Grammatik und des Ausdrucks für den gesamten textuellen Teil der vorliegenden Arbeit angewendet. Dabei wurden alle Vorschläge von Grammarly Pro händisch durch den Autor geprüft und nur vereinzelt übernommen bzw. angepasst. Es wurden keinerlei Inhalte oder wissenschaftliche Beiträge dieser Dissertation durch Grammarly Pro erzeugt.

ChatGPT (https://chatgpt.com/) der Firma Open AI, Inc.: ChatGPT wurde bei der Adaption des Designs und der Formatierung dieser Arbeit mit LaTeX und zugehörigen Bibliotheken sowie für die Erstellung von Grafiken dieser Arbeit mittels Python und Juyper Notebook und zugehörigen Bibliotheken als Lern- und Konfigurationsunterstützung einzelner Bibliotheken verwendet. Die Vorschläge seitens ChatGPT wurden dabei händisch durch den Autor geprüft und vereinzelt übernommen bzw. angepasst. Zudem diente es zum Finden und Wiederauffinden wissenschaftlicher Literatur. Im Konkreten wurde eine Arbeit, die sich mit den unterschiedlichen Lerntypen von APIs beschäftigt gefunden bzw. Literatur für den Beweis der NP-Vollständigkeit des Subgraph-Isomorphismus-Problems ermittelt. Die so gefundene Literatur wurde selbstständig durch den Autor im Original gelesen und auf Validität geprüft. Es wurden keinerlei Inhalte oder wissenschaftliche Beiträge dieser Dissertation durch ChatGPT erzeugt.

Es wurde die Richtlinie Empfehlung zur Nutzung generativer KI in Dissertationen an der Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg (Beschluss des Fakultätsrates 07.2024 – 015/24) beachtet.

Magdeburg, den 06.10.2025

Sebastian Nielebock

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Magdeburg, den 06.10.2025

Sebastian Nielebock