Bachelorstudiengang Angewandte Informatik
Studiengang: BAIN10

Weiterentwicklung von OpenSSL.NET als Schnittstelle zwischen der freien OpenSSL-Bibliothek und dem .NET Framework für den Einsatz in Smart Metering Systemen

Eingereicht von: Erik Groß
Geboren am: 10.08.1991
Matrikel-Nr.: 17717

Erstbetreuer: Prof. Dr. rer. nat. Uwe Heuert
Zweitbetreuer: Dipl.-Ing. (FH) Oliver Punk

Unterschrift Erstbetreuer

Unterschrift Erik Groß

Inhaltsverzeichnis

Abkürzungsverzeichnis

AES	Advanced Encryption Standard
BIO	Binary Input/Output (Abstraktion der Ein-/Ausgabe)
BSD	Berkeley Software Distribution
BSI	Bundesamt für Sicherheit in der Informationstechnik
CA	Certificate Authority
CIL	Common Intermediate Language
CMS	Cryptographic Message Syntax
CN	Common Name
DER	Distinguished Encoding Rules
DH	Diffie-Hellman (Schlüsselaustauschverfahren)
DLL	Dynamic Link Library
DSA	Digital Signature Algorithm
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDHE	Elliptic Curve Diffie-Hellman Ephemeral (Folgenlosigkeit)
ECDSA	Elliptic Curve Digital Signature Algorithm
HAN	Home Area Network
HTTP	Hypertext Transfer Protocol
IPSec	Internet Protocol Security
LMN	Local Metrological Network
MD	Message Digest
NIST	National Institute of Standards and Technology
PEM	Privacy Enhanced Mail
PKI	Public Key Infrastruktur
RFC	Request for Comments
RSA	asymmetrisches kryptographisches Verfahren, benannt nach Ronald L. Rivest, Adi Shamir und Leonard Adleman
SHA	Secure Hash Algorithm
SMGW	Smart Meter Gateway
S/MIME	Secure/Multipurpose Internet Mail Extensions
SSL	Secure Sockets Layer
TLS	Transport Layer Security
VB	Visual Basic (Programmiersprache)
WAN	Wide Area Network

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Zukunft des Mess- und Zählerwesens heißt *Smart Metering*. Beim Smart Metering werden intelligente Zähler verwendet, die den Stromverbrauch der Kunden erfassen und diesen in Echtzeit an den Stromversorger übertragen können. Auf diese Weise kann der Stromversorger sich auf ändernde Netzbelastungen schneller einstellen und dementsprechend reagieren. Die Hochschule Merseburg hat es sich zur Aufgabe gemacht, funktionierende Prototypen eines solchen intelligenten Systems zu entwickeln. Die Komponenten sollen vorzugsweise mit dem *.NET*[1] *Framework* implementiert werden. Für die gesicherte Netzwerkkommunikation und Daten werden kryptographische Verfahren und Objekte benötigt, welches dieses Framework derzeit nicht bietet. Die freie Open Source Bibliothek *OpenSSL* unterstützt nach einer Erweiterung die notwendigen Algorithmen, Objekte und Protokolle. Im Rahmen der Arbeit soll die Schnittstellenbibliothek *OpenSSL.NET* weiterentwickelt werden, die die OpenSSL-Funktionalität in .NET zur Verfügung stellt.

1.1 Aufbau der Arbeit

In dieser Arbeit werden zunächst Funktionen vom .NET Framework, OpenSSL und OpenSSL.NET erläutert. Im zweiten Teil wird der Umfang von OpenSSL.NET betrachtet. Im letzten Teil wird die Weiterentwicklung des OpenSSL.NET Projekts beschrieben, wobei das Schema verfolgt wird, dass zunächst immer ein bestimmtes Problem beschrieben wird, woraufhin Lösungsmöglichkeiten vorgestellt und erläutert werden. Von diesen Möglichkeiten wird eine als die bevorzugte gewählt, wobei diese Wahl begründet wird. Zur Formatierung ist zu sagen, dass Fachbegriffe und markante Stellen *kursiv* dargestellt werden. Erscheinen die Begriffe häufiger, geschieht dies nur beim ersten Vorkommen. Textteile und Namensgebungen, die sich direkt auf einen Quellcode beziehen, werden in der von Visual Studio üblichen Formatierung dargestellt. Größere Quelltextausschnitte werden eingerahmt, der Text davor bzw. danach bezieht sich direkt auf diese Ausschnitte.

1.2 Entwicklersystem

Für die Entwicklung wurden zwei verschiedene Systeme benutzt, die vom Aufbau her ähnlich waren. Das Hauptsystem läuft mit einem 32 Bit Windows 8 Betriebssystem. Im System ist ein Intel Core 2 Quad Q8300 Prozessor mit 4 GB Arbeitsspeicher verbaut. Als Entwicklungsumgebung wird Microsoft Visual Studio 2012 benutzt. Das Zweitsystem ist ein 64 Bit Windows 8 Rechner mit einem AMD Phenom II x64 955 Prozessor und 8 GB Arbeitsspeicher. Als Entwicklungsumgebung wird hier die aktuelle Betaversion von Visual Studio 2013 eingesetzt. Beide Systeme nutzen für die Interoperabilität die gleichen Binärdateien von OpenSSL 1.0.1e als Grundlage für Client-Server-Tests und für die Verwendung im OpenSSL.NET. Die Betriebssysteme werden mit Hilfe von automatischen Updates ständig auf dem aktuellen Stand gehalten. In der Tabelle 1-1 sind die wichtigsten Komponenten beider Systeme in einer Übersicht zusammengefasst. Die Entwicklung wird hauptsächlich am *Entwicklersystem A* stattfinden.

[1] .NET wird gesprochen wie „dot net“.

Komponente	Entwicklersystem A	Entwicklersystem B
Prozessor	Intel Core 2 Quad Q8300	AMD Phenom II x64 955
Arbeitsspeicher	4 GB	8 GB
Architektur	32 Bit	64 Bit
Betriebssystem	Windows 8	Windows 8
Entwicklungsumgebung	Visual Studio 2012	Visual Studio 2013
OpenSSL Version	OpenSSL 1.0.1e	OpenSSL 1.0.1e

Tabelle 1-1 Übersicht über Systemkomponenten

1.3 .NET Framework

Von Microsoft gibt es das .NET Framework, welches es ermöglicht Projekte zu erstellen, die einerseits in mehreren Programmiersprachen programmiert werden können und andererseits prozessorunabhängig laufen. Auf diese Weise kann zum Beispiel eine Bibliothek mit der Sprache *C#* entwickeln. Ein anderes Projekt, welches mit *Visual Basic* (VB) entwickelt wird, kann aber auf die Bibliothek zugreifen und dessen Klassen und Methoden verwenden, als wären diese in VB entwickelt worden. Dieses Verhalten wird dadurch erreicht, da die Programme zunächst in eine gemeinsame Zwischensprache, welche *Common Intermediate Language* oder kurz CIL genannt wird, übersetzt werden. Erst zur Laufzeit wird die CIL in den entsprechenden Maschinencode übersetzt. Das originale .NET Framework gibt es ausschließlich für Microsoft Windows Systeme[2].

Damit auf eine in .NET geschriebene Bibliothek verwiesen werden kann, müssen die *Zielframeworks* der Bibliothek und des zu entwickelnden Programms zueinander kompatibel sein. Beim Kompilieren der Programme kann ausgewählt werden, ob der Zielprozessor der 32 oder 64 Bit, oder seit neustem, der ARM Architektur entspricht. Außerdem bietet .NET hier eine weitere Möglichkeit „Any CPU“, bei der das Programm prozessorunabhängig kompiliert und zur Laufzeit die korrekte Version gestartet wird. Bei den meisten Projekten wird „Any CPU“ gewählt, da es prinzipbedingt die größte Flexibilität bietet. Hierbei wird es zu Problemen kommen, wenn es darum geht, auf native Bibliotheken zuzugreifen. Diese werden nur für einen bestimmten Prozessortyp kompiliert. Gibt es eine Auswahl zwischen 32 und 64 Bit, existieren auch zwei verschiedene Dateien zwischen denen für die Verwendung entschieden werden muss. Aus einem 64 Bit Programm kann aufgrund von Inkompatibilitäten nicht auf eine 32 Bit Bibliothek zugreifen, et vice versa. Wird ein 64 Bit Betriebssystem benutzt, so wird eine „Any CPU“ Anwendung im 64 Bit-Modus ausgeführt. Findet ein Zugriff aus diesem auf eine 32 Bit Bibliothek statt, führt das zum Absturz der Anwendung. Um dies zu verhindern, muss beim Projekt explizit 32 Bit als Zielprozessor ausgewählt werden.

Das .NET Framework ist sehr umfangreich und bietet daher auch fast alle Funktionen, die zum Programmieren benötigt werden. Es werden auch Klassen und Methoden bereitgestellt, die die Möglichkeit bieten robuste Client-Server Systeme aufzubauen. Doch werden für das Smart Metering Projekt bestimmte elliptische Kurven, die später genauer erörtert werden, benötigt. Die Netzwerkfunktionalität von .NET Framework in der aktuellen Version 4.5 unterstützt diese nicht.

1.4 OpenSSL

OpenSSL ist ein freies *Open Source Toolkit*, welches dank einer BSD-Artigen Lizenz **/OSSLL/** die Nutzung sowohl für kommerzielle, als auch nicht kommerzielle Projekte gestattet.

[2] Eine plattformübergreifende Alternative ist das Open Source Projekt Mono. **/MON13/**

OpenSSL löste Ende 1998 *SSLeay*, das von Eric A. Young und Tim J. Hudson entwickelt wurde, ab. Dieses Projekt wird nun von Freiwilligen weiterentwickelt. **/OSSLA/** Der Umfang von OpenSSL wird durch eine komplexe *Crypto-Bibliothek*, einer *SSL-Bibliothek* und einem Kommandozeilentool bestimmt. Die Crypto-Bibliothek bietet die verschiedensten Möglichkeiten zur Verschlüsselung und Signierung. Die SSL-Bibliothek ermöglicht Netzwerkkommunikation über SSL und TLS. Das Kommandozeilentool bietet einen schnellen Zugang zu den wichtigsten Funktionen der Bibliotheken.

Nachdem SSLeay mit der Version 0.9.1b eingestellt worden war, wurde OpenSSL mithilfe dessen Quellcodes weiterentwickelt und am 28. Dezember 1998 als Version 0.9.1c veröffentlicht. Das Projekt wird seitdem stetig fortgeführt und mehrmals im Jahr aktualisiert. Zum jetzigen Zeitpunkt ist die aktuelle Version die 1.0.1e vom 11. Februar 2013. Auf der Webseite des Projekts ist jede veröffentlichte Version des Quelltextes herunterladbar. **/OSSLS/**

Die Projekthomepage selbst verlinkt nur den in der Programmiersprache C geschriebenen Quellcode, welchen man dann für sein System spezifisch kompilieren kann. Alternativ werden für Microsoft Windows Betriebssysteme auf eine externe Seite verwiesen, auf der die bereits kompilierten Binärdateien als Installer zum Download bereitstehen. Neben der aktuellen Version 1.0.1e werden auch die Versionen 1.0.0k und 0.9.8y angeboten. **/SLP13/**

OpenSSL bietet die Nutzung von elliptischen Kurven in der gesicherten Netzwerkkommunikation. Man kann OpenSSL aber nicht bequem mit dem .NET verwenden, da es im nativen Code und nicht im CIL vorliegt. Somit ist die Bibliothek *unmanaged*[3]. Bei der Verwendung muss stets darauf geachtet werden, dass der Speicher, der reserviert und verwendet wurde, auch wieder freigegeben werden muss. Um dies zu vereinfachen gibt es ebenfalls ein Open Source Projekt namens „OpenSSL.NET“.

1.5 OpenSSL.NET

OpenSSL.NET wurde unter der gleichen Lizenz wie OpenSSL veröffentlicht **/ONETL/** . Das Ziel dieses Projekt ist es, die Vorteile der objektorientierten .NET-Welt mit der schnellen nativen OpenSSL Bibliothek zu vereinen. OpenSSL.NET ist ein *Wrapper*, der den Zugang zu den Funktionen der OpenSSL Bibliothek in eine private Klasse kapselt und den Zugriff nach außen über andere spezielle Klassen gewährt.

In OpenSSL werden Strukturen verwendet, die in den Funktionen befüllt werden. Ein *Pointer*[4] zu dieser erstellten Struktur wird zurückgegeben. Dieser Pointer muss beim Verwenden anderer Funktionen immer mit übergeben werden.

OpenSSL.NET handhabt es so, dass in einer Klasse eine Methode aufgerufen wird. Diese Methode greift auf die native dynamische Programmbibliothek (DLL) zu, und speichert den zurückgegeben Pointer in eine Instanzvariable. Bei anderen Funktionen, die auf diesen Pointer angewiesen sind, wird dieser mit übergeben. Der Entwickler muss nicht selbst für diese Übergabe sorgen.

[3] .NET Anwendungen sind *managed*, das heißt der Speicher dieser wird verwaltet und automatisch wieder frei gegeben, sobald er nicht mehr benötigt wird. Bei *unmanaged*(nicht verwalteten) Anwendungen muss der allokierte Speicher immer manuell wieder dealloziert werden.

[4] Ein Pointer ist ein Zeiger auf eine Speicheradresse im Arbeitsspeicher. In diesem Fall zeigt der Pointer auf die Startadresse der befüllten Struktur.

Dieses Projekt wurde im März 2006 ins Leben gerufen, die aktuelle Version ist die Version 0.5 von 29. März 2012 **/ONETCC/** .

2 Analyse von OpenSSL.NET

Zunächst muss geprüft, auf welchen Stand dieses Projekt ist. Sowohl aus der *Readme*[5], als auch aus dem Quellcode selbst kann entnommen werden, dass das OpenSSL.NET auf Version 1.0.0d (Februar 2011) von OpenSSL basiert und für das .NET Framework in der Version 2.0 geschrieben wurde. Im Laufe der Arbeiten soll der Wrapper auf der OpenSSL Version 1.0.1e umgestellt werden, um so auf dem aktuellen Stand zu sein, was vor allem der Sicherheit zugunsten kommt. Weiterhin wird das .NET Framework auf die aktuelle Version 4.5 umgestellt, damit die Vorteile von Kompatibilität, Sicherheit und neuer Sprachversionen genutzt werden können. Das .NET Framework 2.0 wurde im Jahr 2006 zur Verfügung gestellt und der *Mainstream-Support* lief bereits im Jahr 2011 ab **/MSSLN2/** .

Die Version 2.0 des Frameworks setzt unter anderem auch auf die Sprachversion 3.0 von C#, welche viele Vorteile der aktuellen Sprachversion 5.0, die ab dem .NET Framework 4.5 genutzt werden kann, nicht bietet. Einige dieser Vorteile sind unter anderem die Verwendung einfacher asynchroner Methoden. Weiterhin gibt es die Möglichkeit optionale Parameter direkt mit ihrem Namen zu setzen. Eine weitere praktische Neuerung ist die mit Version 4.0 eingeführte Funktion der dynamischen Bindung. Es kann über das `dynamic`-Schlüsselwort festlegen werden, dass die Überprüfung des Datentypen zur Kompilierzeit ignoriert wird. Dadurch lässt sich zur Entwicklungszeit jede beliebige Operation ausführen. Das wiederum hat den Vorteil, dass sich beliebige Objekte zusammenbauen und nutzen lassen. Andererseits beherbergt es die Gefahr, dass unerwartete Laufzeitfehler auftreten können, wenn die Typen inkompatibel sind. **/MSDNNET45/**

Als Entwicklungsumgebung wird das Visual Studio von Microsoft in der 2012er Version benutzt. Nachdem der bestehende Quellcode von OpenSSL.NET heruntergeladen und geöffnet wurde, ist der erste Test, ob sich das Projekt fehlerfrei kompilieren lässt. Das Kompilieren wurde erfolgreich beendet. In der Projektmappe befinden sich drei Projekte:

- Ein Klassenbibliotheksprojekt für den Wrapper selbst, welches in andere Projekte als Abhängigkeit eingebunden werden kann.
- Ein weiteres Projekt für eine Konsolenanwendung, die eine ähnliche Funktionalität der Konsolenanwendung der nativen Bibliothek bereitstellen soll,
- und ein drittes Projekt welches für Komponententests gedacht ist.

Im folgendem werden diese drei Projekte untersucht und bewertet.

2.1 Das Wrapper Projekt

Das Wrapper-Projekt trägt den Namen „ManagedOpenSsl-2010“ und besteht aus vier Teilen. Es gibt den „Core“ in dem gemeinsame Klassen und Methoden untergebracht sind, dazu gehört die Klasse für die Verwaltung der nativen Funktionen, eine weitere Klasse für die binären Ein- und Ausgaben, welche im OpenSSL-Umfeld kurz *BIO* genannt werden. Wichtig

[5] Eine Readme ist eine Datei, die meist bei Softwarepaketen beiliegt und kurz erläutert, was das für eine Software es ist und wie sie verwendet wird.

zu erwähnen ist hier auch die Base-Klasse, welche die Basisklasse für die OpenSSL Objekte darstellt. Sie verwaltet den Umgang mit den Pointern für die nativen Strukturen.

Ein weiterer Teil „SSL“ ist den verschlüsselten Netzwerkverkehr gewidmet. Hier befinden sich Klassen für die Erstellung von *Streams* sowohl für die Client- als auch für die Serverseite. Es gibt eine Klasse „`SslStream`“, welche die Schnittstelle zwischen der Bibliothek und dem Framework herstellt, indem es die `AuthenticatedStream`-Klasse implementiert. Dies wiederum ermöglicht es die OpenSSL Netzwerkfunktionalitäten mit denen des Frameworks, wozu unter anderem der `TcpListener` gehört, zu verwenden. Der `SslStream` verweist auf die „`SslStreamBase`“ Klasse, welche die Basisklasse für „`SslStreamClient`“ und „`SslStreamServer`“, die die Initialisierung und den Verbindungs-*Handshake* für den Client bzw. Server übernehmen, darstellt. Die Basisklasse übernimmt das Lesen und Schreiben vom und in den Stream über die nativen OpenSSL Funktionen. Damit eine Verbindung entstehen kann wird ein „`SslContext`“ benötigt, der über die `SslStreamClient` bzw. `SslStreamServer` Klasse initialisiert wird. Der Kontext entscheidet welche „`SslCipher`“ und welche „`SslMethod`“ benutzt werden. Eine letzte „`Ssl`“ Klasse verwaltet Funktionen zum Annehmen und Schließen von Verbindungen, außerdem übergibt diese Klasse die zu verwenden Zertifikate an die native Bibliothek. Die komplexen Beziehungen dieser SSL-Klassen sind in der Abbildung 2.1 noch einmal stark vereinfacht dargestellt. Ein vollständiger Beziehungsüberblick würde der Übersicht an dieser Stelle schaden.

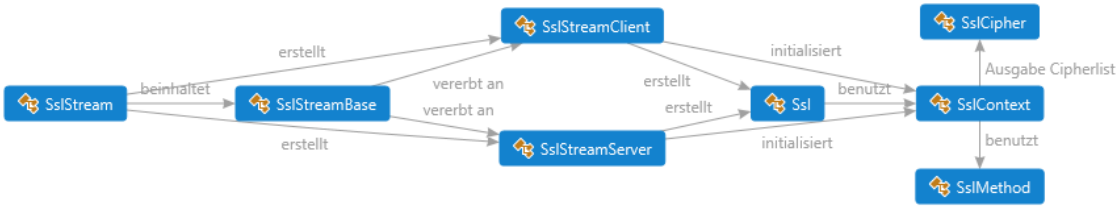

Abbildung 2.1 Stark vereinfachte Beziehung zwischen den SSL-Klassen

Beim Analysieren der SSL-Klassen ist aufgefallen, dass viele Funktionen auskommentiert wurden und einige *Workarounds* benutzt wurden. So wurde statt dem expliziten einschränken der SSL Optionen sämtliche *Workarounds* gegen ehemalige *Sicherheitspatches* aktiviert. Diese wurden in OpenSSL eingebaut, um Kompatibilitäten zu älteren OpenSSL versuchen aufrecht zu erhalten, bei Grund auf neuentwickelten Anwendung sollten diese nicht verwendet werden. Weiterhin ist auffällig, dass ausschließlich die „SSLv23_client_method“ bzw. „SSLv23_server_method“ Funktionen aktiviert wurden, welche die gleichzeitige Verwendung von SSL in der Version 2 und 3, sowie TLS 1.0 ermöglichen, obwohl andere Funktionen für die Verwendung einer spezifischen Version in OpenSSL.NET implementiert sind. Es fehlt aber die Implementierung von TLS 1.1 und TLS 1.2.

Der dritte Teil „X509“ beinhaltet, wie es der Name schon vermuten lässt, sämtliche Klassen, die für die Verwaltung und Erstellung von Zertifikaten notwendig sind. Es gibt die Möglichkeit *Certificate Authorities* zu erstellen und mit diesen aus *Zertifikatrequests* Zertifikate zu erstellen und diese zu signieren. Es wird auch die Möglichkeit geboten, die Zertifikate mittels *X.509 Extensions* die Zertifikate zu erweitern, um so zum Beispiel den Verwendungszweck festzulegen.

Der vierte und letzte Teil des ManagedOpenSsl-Projekts nennt sich „Crypto“. Hier sind Klassen und Methoden für kryptographische Verfahren, welche Ver- und Entschlüsselung

und Signierung beinhalten, untergebracht. Ein der wichtigsten Klassen ist hier die `CryptoKey`-Klasse, welche die verschiedenen Algorithmen in einer Klasse vereint und so eine Allgemeine Zugriffstelle bietet, um Schlüssel zu generieren und private und öffentliche Schlüssel zu speichern und zu verwenden. Unterstützt werden sowohl RSA-Schlüssel, DSA-Schlüssel, Diffie-Hellman Schlüssel, als auch verschiedene Elliptische Kurven. Zu den Unterstützen Kurven gehören *NIST prime*, *X9.62* Kurven sowie eine *IPSec* Kurve. Vom nativen OpenSSL werden 65 verschiedene elliptische Kurven unterstützt, davon sind in OpenSSL.NET nur 24 implementiert. Weiterhin sind *HMAC*, eine Methode für die Nachrichtenauthentifizierung mittels *Hash*, sowieso verschiedene Einweg Hash Verfahren, wie *MD5*, *SHA-1* oder *SHA-2*, implementiert.

2.2 Das Kommandozeilen-Projekt

Das Kommandozeilentool des nativen OpenSSL ist ein sehr starkes Werkzeug, wenn es darum geht schnell, ohne Programmieraufwand Ergebnisse zu erreichen. Es wird die Möglichkeit geboten Schlüssel mit verschiedensten Verfahren, Zertifikate und *Zertifikatrequests* zu erstellen, des weiteren können Zertifikate auf ihre Gültigkeit überprüft oder codierte Daten aufgeschlüsselt angezeigt werden. Auch sehr praktisch ist die Funktion zum Erstellen eines Servers oder Clients, damit verschlüsselter Netzwerkverkehr auf seine Funktionalität geprüft werden kann.

Von der ganzen Palette der möglichen Befehle ist zurzeit in OpenSSL.NET nur ein Bruchteil im Projekt „cli-2010“ implementiert. Dazu gehört der „dh“ Befehl, welcher die Diffie Hellman Parameter aus einer Datei im *PEM* oder *DER* Format liest, diesen kontrolliert und/oder in einem anderen Format wieder abspeichert. Weiterhin ist der „gendh“ Befehl implementiert, welcher DH-Parameter generiert. Es existieren mit „rsa“ und „genrsa“ äquivalente Versionen für den RSA-Algorithmus, wobei anzumerken ist, dass bei genrsa ein Standardwert von 512 Bit benutzt wird, welcher heutzutage als nicht mehr sicher einzuschätzen ist. Der „version“ Befehl bietet die Möglichkeit die Version, das Erstelldatum und weitere Eigenschaften anzuzeigen. Sowohl der Befehl „enc“, als auch „dgst“ sind unvollständig implementiert. Diese sollten dem de- und encodieren von Dateien bzw. dem Signieren und Prüfen von Signaturen dienen. Alle anderen Funktionen, die es zum Beispiel ermöglichen Zertifikate zu erstellen, oder zu prüfen, oder Schlüssel mittels elliptischen Kurven zu generieren, sind nicht implementiert.

2.3 Das Test-Projekt

Das dritte Projekt „UnitTests-2010“ in der Projektmappe ist ein Komponententestprojekt. Mit Komponententests kann man bestimmte Teile seiner Software auf korrekte Funktionalität testen. In der Regel wird ein fester Erwartungswert hinterlegt. Nachdem eine Methode ausgeführt wurde, wird dessen Rückgabewert mit dem Erwartungswert verglichen. Ist der Rückgabewert mit dem Erwartungswert identisch, gilt der Test als erfolgreich. Die Prüfung findet mit Hilfe von *Assertionen*[6] statt. Eine weitere genutzte Möglichkeit für die Tests ist das Prüfen auf erwartete Ausnahmen, das heißt es wird eine Methode ausgeführt, die auf einen Fehler stößt und eine Ausnahme auslöst. Die Testmethode kontrolliert darauf, ob eine Ausnahme ausgelöst wurde und ob es die Erwartete war.

[6] Das Wort „Assertion“ kommt ursprünglich aus dem Lateinischen und bedeutet so viel wie Behauptung. Mit einer Assertion wird eine Behauptung, welche dem Erwartungswert entspricht, aufgestellt und mit der Realität verglichen.

Das Test-Projekt von OpenSSL.NET setzt auf das *NUnit* Framework, ein Open Source Komponententest-Framework. Der Nachteil von *NUnit* ist, dass ein *Build* benötigt wird, das auf der gleichen .NET Framework Version basiert, wie das zu testende Projekt. Da geplant ist OpenSSL.NET von der Framework-Version 2.0 auf Version 4.5 umzustellen, wird eine Kopie vom Test-Projekt angelegt und dieses auf *Visual Studio Unit Testing Framework*, welches in Visual Studio Professional seit der 2005er Version einen festen Bestandteil darstellt. Andernfalls würden zwei Versionen von *NUnit* benötigt werden, eine die mit dem .NET Framework 2.0 kompiliert wurde und eine weitere für das .NET Framework 4.5.

Die Umstellung des *Testframeworks* ist aufgrund des einheitlichen Vorgehens bei Komponententests einfach zu realisieren. Testklassen werden im *NUnit* Framework mit dem Attribut „`[TestFixture]`“ markiert, im *Visual Studio Unit Testing Framework* werden diese mit dem Attribut „`[TestClass]`“ markiert. Damit beide Frameworks die passenden Testmethoden finden können, müssen diese außerdem mit einem Attribut markiert werden. Im *NUnit* wird „`[Test]`“ verwendet und im *Visual Studio Unit Testing Framework* wird es mittels „`[TestMethod]`“ markiert. Für Tests, die übersprungen werden sollen, verwenden beide Frameworks das „`[Ignore]`“ Attribut. Es muss also keine Änderung stattfinden. Für Assertionen verwenden die `Assert`-Klassen des jeweiligen Frameworks verwendet. Beide Klassen haben ähnliche Methoden mit den gleichen Parametern. Daher müssen auch kaum Anpassungen gemacht werden. Ausnahmen stellen dabei Befehle wie „`Assert.Greater(aout, 4);`“ dar, welche es nur in *NUnit* gibt. Diese müssen durch entsprechende Alternativen des *Visual Studio Unit Testing Frameworks* wie „`Assert.IsTrue(aout > 4);`“ ersetzt werden. Bevor die Tests ausgeführt werden können, muss für das OpenSSL.NET Projekt die `TestBase`-Klasse angepasst werden. Diese Klasse übernimmt das Starten und Beenden des `MemoryTracker`s, welcher den Speicher der nativen OpenSSL Bibliothek überwacht und eventuelle Speicherlecks, die häufig als Memoryleak bezeichnet werden, aufdeckt. Die Klasse wird als Basisklasse für jede Testklasse festgelegt. Die `TestBase`-Klasse besitzt eine Methode „`Setup`“, welche „`[SetUp]`“ markiert ist und den Memory-Tracker startet, und eine Methode „`Teardown`“, welche mit „`[TearDown]`“ markiert ist und den Memory-Tracker beendet. Im Visual Studio Unit Testing Framework wird das Attribut der `Setup`-Methode durch das äquivalente „`[TestInitialize]`“ und das der `Teardown`-Methode durch „`[TestCleanup]`“ ersetzt.

Abbildung 2.2 Entladenes Projekt

Das original „UnitTests-2010“-Projekt kann nun entladen werden, das heißt dessen Code wird nicht mehr von Visual Studio verwaltet, analysiert und kompiliert, bis das Projekt erneut geladen wird. Diese Maßnahme verhindert zum einem Verwechselungen, da das Projekt ein anderes Symbol bekommt (Abbildung 2.2), und zum anderen verringert das Entladen von unnötigen Projekten die Zeit, die für das Öffnen und auch Kompilieren der Projektmappe benötigt wird. Nun können die Tests über den integrierten Test-Explorer ausgeführt werden. Die Tests können entweder einzeln, in einer Gruppe oder alle gemeinsam ausgeführt werden. Wurden alle ausgewählten Tests ausgeführt, kann eine Zusammenfassung – wie in Abbildung 2.3 sichtbar – in der aufgelistet ist, wieviele Testfälle erfolgreich durchgeführt wurden. Erfreulicherweise wurden die meisten vorhanden Tests, die nicht als zu ignorieren markiert wurden,

Abbildung 2.3 Zusammenfassung Testergebnisse

erfolgreich abgeschlossen. Getestet werden verschiedene Hashalgortihmen, Schlüsselgeneratoren und unterschiedliche Methoden zu Zertifikaten.

Der Testfall „TestRSA“, welcher Ver- und Entschlüsselung, sowie Tests zu *Optimal Asymmetric Encryption Padding* (OAEP) beinhaltet, nimmt auf dem Entwicklungssystem A (siehe Tabelle 1-1) eine große Laufzeit in Anspruch und wird aus diesem Grund zunächst manuell abgebrochen. Ein späterer Test auf dem anderen System zeigt, dass dieser erfolgreich abgeschlossen wird. Die einzigen Tests, welche scheiterten, sind „CanGetAsPEM“ aus der `TestX509Certificate`-Klasse und „TestCaseServer“ aus der `TestServer`-Klasse. Der erste Testfall lädt ein Zertifikat im PEM Format, speichert dies als Erwartungswert ab und erstellt gleichzeitig mittels diesen Strings ein Zertifikatobjekt, welches ausgegeben und mit dem Erwartungswert verglichen wird. Die beiden Strings sind bei genauerer Analyse identisch. Einzig der Zeileneinzug zweier Eigenschaften stimmt nicht überein. Dies kann Aufgrund unterschiedlicher *OpenSSL-Builds* zurückgeführt werden, in denen der Einzug anders gehandhabt wird. Der letzte Test, der fehlschlägt, der auch im weiteren Verlauf eine Rolle spielen wird, ist das Testen eines Client-Server-Systems sowohl im synchronen, als auch asynchronen Modus. Für beide Modi werden drei unterschiedliche Verbindungsarten getestet:

- *Basic Test*: Dieser Test testet eine Verbindung, die nur den Server über ein Zertifikat authentifiziert. Es wird das Standardprotokoll benutzt, welches sowohl SSLv2, SSLv3 als auch TLSv1 zulässt mit einer mittleren Verschlüsslungstärke, was die Verwendung einer 128 Bit Verschlüsselung impliziert.
- *Intermediate Test*: Dieser Test testet ebenfalls eine Verbindung, die nur den Server über ein Zertifikat authentifiziert. Es wird TLSv1 mit einer beliebig starken Verschlüsselung benutzt.
- *Advanced Test*: Bei diesem Test wird sowohl eine Server-, als auch eine Clientauthentifizierung über Zertifikate durchgeführt. Genau wie beim IntermediateServerTest wird hier TLSv1 mit einer beliebig starken Verschlüsselung benutzt.

Einzig der *Basic Test* wird erfolgreich ausgeführt. Für die beiden anderen Tests wurden einige Methoden noch nicht vollständig ausprogrammiert, was im späteren Teil noch nachgeholt wird. Wären die beiden Tests erfolgreich, gäbe es aber dennoch ein weiteres Problem mit den Test selbst: Die Kontrolle, ob die richtige Nachricht empfangen wurde, ist schlicht falsch.

```
if (String.Compare(serverReadBuffer.ToString(),
                   clientMessage.ToString()) != 0)
{
   Console.WriteLine("BasicServerTest Read Failure:\nExpected:{0}\nGot:{1}",
                     clientMessage.ToString(), serverReadBuffer.ToString());
   ret = false;
}
```

In diesem Codeausschnitt werden zwei Variablen verglichen. Sowohl `serverReadBuffer` als auch `clientMessage` sind vom Typ ein Byte-Array, welches selbst keine Überladung für die `ToString`-Methode besitzt. Das heißt im Umkehrschluss, wenn `ToString` ausgeführt wird, dass die vererbte Methode von der übergeordneten `Object`-Klasse ausgeführt wird. Diese gibt allerdings nur eine Zeichenfolge, die den Typen beschreibt, zurück. In diesem Fall ist es „System.byte[]“. Diese Zeichenfolge ist für beide Variablen immer die gleiche, weshalb

dieser Vergleich niemals negativ ausfallen wird. Dieser grobe Fehler wird sofort ausgebessert und durch die folgende Zeile ersetzt:

```
Assert.AreEqual(Encoding.ASCII.GetString(serverMessage),
                Encoding.ASCII.GetString(clientReadBuffer).Trim('\0'));
```

Diese Zeile wandelt beide Arrays in Strings um, wobei davon ausgegangen wird, dass das Byte-Array *ASCII-Kodiert* ist. Der vom Clienten empfange Vergleichswert, muss gekürzt werden, da nicht benutzte Stellen des Arrays mit Nullen gefüllt werden. Eine Null entspricht im ASCII-Code der maskierten Zeichenkette „\0“, wodurch die Zeichenkette länger und damit ungleich dem Vergleichswert ist. Ein erneuter Test nach dieser Korrektur zeigt, dass der *Basic Test* nach wie vor erfolgreich abgeschlossen wird. Ein *Schritt-für-Schritt-Debugging* kann die korrekte Übertragung ebenfalls bestätigen. In der Abbildung 2.4 ist im „Überwachen 1“-Fenster die Originaldaten zu sehen, die vom Client gesendet werden, im „Überwachen 2“-Fenster ist die vom Server empfangene Nachricht zu erkennen. Werden beide verglichen, ist erkennbar, dass die Daten gleich sind. Anstatt dem `Trim('\0')` könnte auch der Rückgabewert der `Read`-Methode benutzt werden, der genau die Anzahl der empfangenen Bytes angibt. Da dies aber nur ein Test für Serverfunktionalität ist, und ausschließlich Texte und keine Binärdaten übertragen werden, ist die `Trim`-Methode in diesem Fall ausreichend.

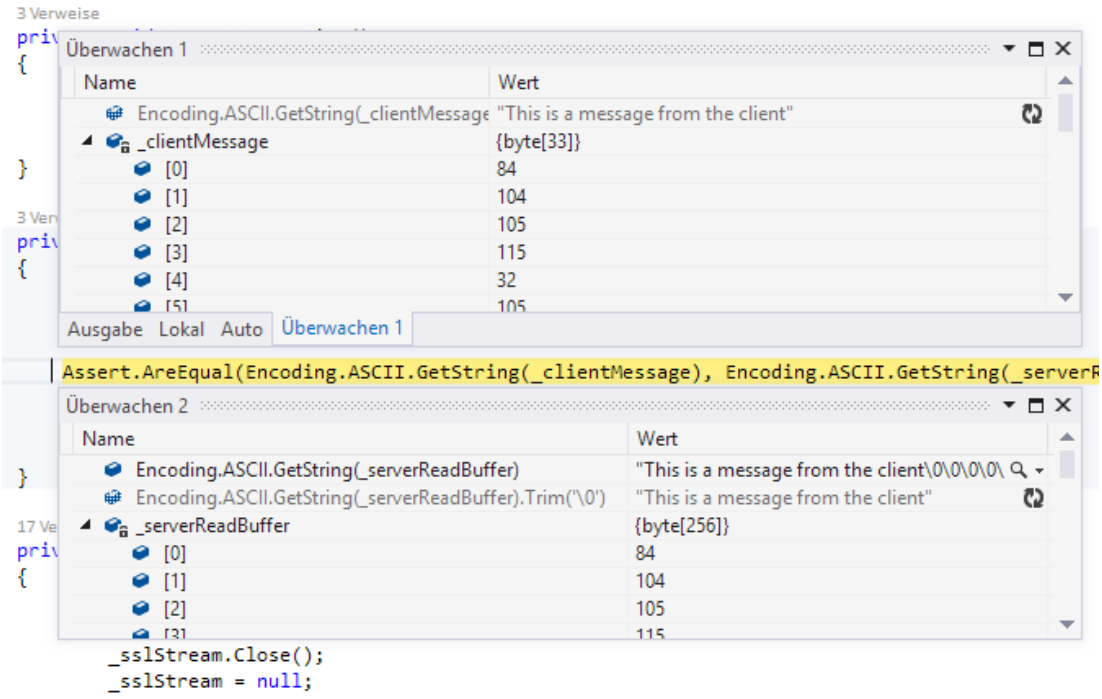

Abbildung 2.4 Debug-Ansicht Basic Test

3 Smart Metering

In Deutschland spielen erneuerbare Energien eine immer wichtigere Rolle. In den letzten Jahren gab es einen stetigen Zuwachs beim Gewinnung und Verbrauch natürlicher Energien, die zum Beispiel durch Wind-, Wasserkraft oder Photovoltaikanlagen gewonnen werden.

Gleichzeitig gibt es seit der Nuklearkatastrophe in Fukushima 2011 einen massiven Rückgang in der Nutzung von Kernenergie. **/AGE13/** Es ist sehr wahrscheinlich, dass der Trend des steigenden Anteils von erneuerbaren Energien anhält. Unser Stromnetz ist von einer konstanten Frequenz von 50Hz abhängig. Wird diese nicht gehalten, funktionieren die Geräte, die für diese 50Hz gebaut wurden, nicht mehr. Im Extremfall könnte dies zu einem kompletten *Blackout* führen. Um diese Frequenz halten zu können, muss genauso viel Energie ins Stromnetz eingespeist werden, wie genutzt wird. Ist der Bedarf höher als das Angebot, sinkt die Frequenz, et vice versa. Die erneuerbaren Energien liefern ungleichmäßig Strom, da die Natur nicht beeinflussbar ist. Mit den herkömmlichen Energiequellen war es kein Problem auf den ändernden Verbrauch, der sich durch eine steigende bzw. sinkende Frequenz auswirkt, zu reagieren und die Stromerzeugung entsprechend anzupassen. Durch den Zuwachs der erneuerbaren Energiequellen werden Stromschwankungen immer häufiger.

Um das Problem der sich ändernden Netzfrequenz entgegenzuwirken, wird es intelligente Netze, die *Smart-Grids* genannt werden, geben. Solche ermöglichen Strom zu speichern, oder anderweitig zu nutzen, wenn dieser nicht benötigt wird. Zu gegebener Zeit kann der gespeicherte Strom zurück ins Netz geführt werden, wenn der Bedarf wieder steigt. Damit die *Smart-Grids* reagieren können, bevor sich die Frequenz ändert, werden in den Haushalten intelligente Messsysteme, welche als Smart Metering Systems bezeichnet werden, eingesetzt. Diese Stromzähler erfassen den Verbrauch des Haushalts elektronisch und ermöglichen es, den aktuellen Verbrauch in Echtzeit an den Stromversorger über eine Netzwerkverbindung zu übermitteln. Die Verbindung muss aufgrund des Datenschutzes und als Sicherung vor Manipulationen kryptographisch gesichert werden. **/BSISMS13/**

3.1 Anforderungen an Smart Metering

Das *Bundesamt für Sicherheit* (BSI) hat für die intelligenten Messsysteme eine Reihe von technischen Richtlinien, die unter der Bezeichnung „BSI TR-03109“ zusammengefasst werden, erarbeitet. Damit die Smart Meter in Deutschland eingesetzt werden dürfen, müssen zunächst diese Richtlinien erfüllt werden. **/BSITR13/** Die Anforderungen für die kryptographischen Verfahren werden durch die BSI TR-03109-3 „Kryptographische Vorgaben für die Infrastruktur von intelligenten Messsystemen“ festgelegt. In dieser Richtlinie steht, dass die jeweils aktuelle Fassung der BSI TR-03116-3 „eCard-Projekte der Bundesregierung Teil 3“, welche einmal jährlich aktualisiert wird, anzuwenden ist. **/BSITR031093/**

Die Vorgaben, die in diesem Dokument **/BSITR031163/** genannt werden, sind für 6 Jahre ausgelegt. Das bedeutet zum aktuellen Zeitpunkt, dass die Angaben bis 2019 ihre Gültigkeit beibehalten werden. Für digitale Signaturen und Verwendung von Schlüsseln werden zurzeit ausschließlich elliptische Kurven (EC) für die Benutzung vorgeschrieben. Für diese elliptische Kurven müssen verschiedene *Domain-Parameter* unterstützt werden, dazu gehören *BrainpoolP256r1*, *BrainpoolP384r1*, *BrainpoolP512r1*, *NIST-P-256* und *NIST-P-384*.

Des Weiteren wird die Art der Zufallsgeneratoren vorgeschrieben. Es muss ein Zufallsgenerator der Klasse *DRG.3*, *DRG.4*, *PTG.3* oder *NTG.1* verwendet werden. Da der Verwendungszweck von OpenSSL.NET im Moment ausschließlich für Prototypen gedacht ist, werden im Rahmen dieser Arbeit statt Hardware-Zufallszahlengeneratoren lediglich die in OpenSSL integrierten Pseudozufallsgeneratoren verwendet.

Für die Authentifizierung von öffentlichen Schlüsseln in Smart Metering Systemen wird eine *Public Key Infrastruktur* (PKI) verwendet. Wie in Abbildung 3.1 sichtbar ist, besteht diese aus

einem Root-CA als nationale Zertifikatautorität, mehrere Sub-CAs, die vom Root-CA signiert werden, und mehrere Nutzerzertifikate, die wiederum von den Sub-CAs signiert werden. Es wird also beispielsweise jedem Energieversorger ein Sub-CA ausgestellt und dieser stellt seinen Kunden, den Stromabnehmern, und den *Smart Meter Gateways* ein Zertifikat aus. Die Signaturen verwenden ausschließlich den ECDSA, beim Root-CA wird dies mit SHA384 und bei den Sub-CAs mit SHA256 kombiniert. Als EC-Domain-Parameter sind die NIST-P-384 bzw. NIST-P-256 Kurven zu verwenden. Es ist geplant, dass generell die gestatteten *NIST-Kurven* bis Ende 2015 durch entsprechende *Brainpoolkurven* ersetzt werden. Die Gültigkeitsdauer der Zertifikate ist im BSI TR-03109-4 festgelegt, welche für das OpenSSL.NET Projekt selbst nicht beachtet werden muss, da dies eine spezifische Implementierung ist.

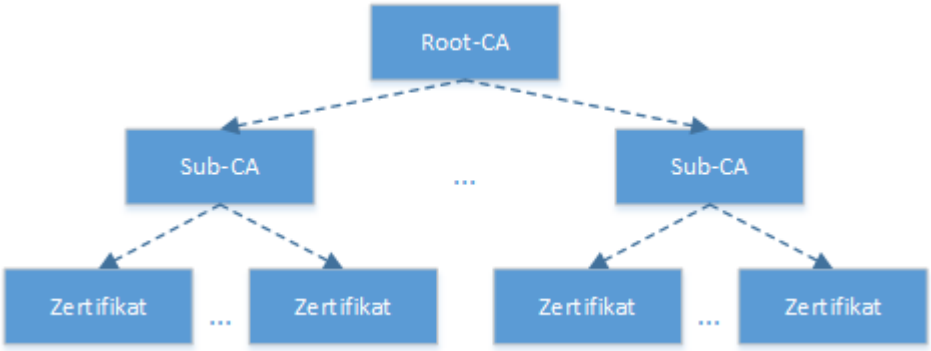

Abbildung 3.1 Public Key Infrastruktur

Die wichtigsten Vorgaben werden zum Thema verschlüsselte Netzwerkkommunikation getätigt. Das Bestehen eines „gegenseitig authentisierten Kanals zwischen Smart-Meter-Gateway und autorisierten Marktteilnehmern“ **/BSITR031163, S. 10/** muss gegeben sein, das heißt, es muss sowohl eine Server- als auch eine Clientauthentifizierung durchgeführt werden. Es ist das TLS Protokoll in der Version 1.2 oder höher zu verwenden und es darf kein *Fallback*[7] auf ältere Versionen geben. Für den Schlüsselaustausch, Hashfunktion, Authentifizierung, Verschlüsselung und *Message Authentication Code*(MAC) dürfen folgende *Cipher Suites* benutzt werden:

- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384

Die *Cipher Suites* beschreiben exakt, welche kryptographischen Verfahren benutzt werden. Die Suite TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 und die NIST-P-256 Kurvenparameter müssen für Verbindungen zwischen dem Smart Meter Gateway und den Endkunden eine Mindestanforderung. Die anderen drei *Cipher Suites* sind mit den anfangs erwähnten elliptischen Kurven zurzeit optional, die Implementierung wird aber für eine zukunftssichere Nutzung empfohlen. Davon abweichende Suiten und EC-Domain-Parameter dürfen nicht verwendet werden. Die für die Authentifizierung notwendigen Zertifikate werden über die NIST Prime-Kurve P-256 und einen dem SHA-256 Hashalgorithmus erstellt.

Das BSI gliedert die Kommunikation in drei Netzwerkbereiche, die in Abbildung 3.2 dargestellt werden. Im Zentrum steht das *Smart Meter Gateway* (SMGW), welches diese

[7] Ein Fallback ist das verwenden eines anderen Verfahrens, falls das erste fehlschlägt.

Bereiche miteinander verbindet. Im *Wide Area Network* (WAN) findet die Kommunikation mit externen Marktteilnehmern und dem SMGW Administrator statt. Der Administrator bekommt vom Anbieter beispielsweise die Anforderung den aktuellen Zählerstand auszulesen. Nachdem er den Vorgang eingeleitet und die nötigen Befehle an das SMGW gesendet hat, antwortet dieses direkt dem externen Marktteilnehmer mit den geforderten Daten. Diese Daten kommen einerseits aus dem *Local Metrological Network* (LMN) und andererseits aus dem *Home Area Network* (HAN). Im LMN sind Zähler für Strom, Gas, Wasser und Wärme, die direkt von den Endverbrauchern gemessen werden. Das HAN grenzt sich durch die Wohnumgebung des Endverbrauchers ab. Es werden Daten mit steuerbaren Energieverbrauchern und Energieerzeugern ausgetauscht. Außerdem erhält dieser weitere Daten vom SMGW, wie zum Beispiel seinen aktuellen Verbrauch oder aktuelle Tarifinfos. Es können auch Daten für Servicetechniker zur Verfügung gestellt werden. **/BSITR031091/**

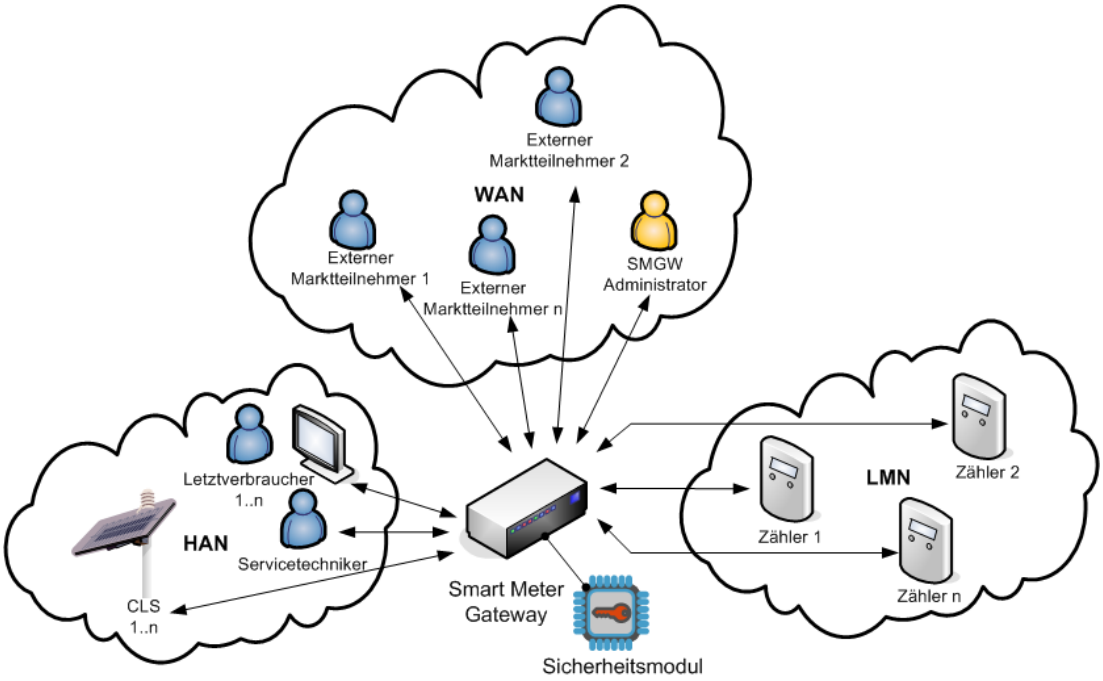

Abbildung 3.2 „Einbettung des Smart Meter Gateways in seine Einsatzumgebung“ ***/BSITR031091, S. 14/***

Für diese unterschiedlichen Netzwerkbereiche gibt es gesonderte Regelungen für die Sitzungsdauer. Eine *Session* im WAN und HAN darf für maximal zwei Tage aktiv sein. Im LMN allerdings darf sie ein Monat gültig sein, wobei aber maximal 5 MB an Daten in einer Session übertragen werden dürfen.

Für die Übertragung der Daten innerhalb einer TLS-Verbindung soll stets eine Ende-zu-Ende-Verschlüsselung benutzt werden. Die verschlüsselten Daten sollen außerdem im *Cryptographic Message Syntax-Format* (CMS) signiert werden. Es muss die Möglichkeit, die Daten mittels AES-GCM zu verschlüsseln und die Authentizität zu sichern, und die Möglichkeit das AES-CBC-CMAC-Verfahren zu benutzen, gegeben sein. Wobei letzteres AES-CBC mit dem Initialvektor Null für die Verschlüsselung und AES-CMAC für die Authentifizierung der Daten verwendet wird. Die Schlüssel für die Nachrichtenauthentifizierungscodes (MAC) müssen vor jeder Verwendung neu generiert und dürfen nur einmalig benutzt werden.

Die Schlüssel sind wiederum verschlüsselt im CMS-Container enthalten. Der Schlüssel für diese Verschlüsselung wird mittels ECKA-EG berechnet. Dabei ist SHA-256 als Hashfunktion und NIST P-256 als Kurvenparameter zu verwenden. Auch hier gilt, dass die NIST prime-Kurve bis 2015 auf Brainpool-Kurven umgestellt werden soll. Mit diesem berechneten Schlüssel kann die Verschlüsselung des ursprünglichen Schlüssels mittels AES Key Wrap Methode stattfinden.

Die verschlüsselten und MAC-gesicherten Daten müssen noch signiert werden. Dazu wird ECDSA mit SHA-256 als Hashfunktion und NIST P-256 als Kurvenparameter genutzt.

Die oben ausgearbeiteten Spezifikationen stammen alle aus der TR 03-116-3. **/BSITR031163/**

3.2 Feature List

Die kryptographischen Anforderungen des BSI an die Infrastruktur des intelligenten Zählerwesens sind nun ausgearbeitet worden. In der folgenden Tabelle 3-1 wird der Status der Implementierung in OpenSSL und OpenSSL.NET gegenübergestellt. Da OpenSSL die Brainpoolkurven noch nicht implementiert hat, hat die Hochschule Merseburg eine Modifikation vorgenommen, die diese Bibliothek erweitert. Zu diesen Modifikationen gehört unter anderem das Hinzufügen von Brainpool in das unterstützte Kurvenrepertoire. In der Tabelle zeigt das grüne Häkchen (✓) an, dass die Funktion implementiert ist. Das rote Kreuz (✗) bedeutet, dass die Funktionalität nicht vorhanden ist.

BSI-Anforderung	OpenSSL	OpenSSL (erweitert)	OpenSSL.NET
Brainpool P256r1	✗	✓	✗
Brainpool P384r1	✗	✓	✗
Brainpool P512r1	✗	✓	✗
NIST P-256	✓	✓	✓
NIST P-384	✓	✓	✓
SHA-256	✓	✓	✓
SHA-384	✓	✓	✓
ECDSA	✓	✓	✓
CMS	✓	✓	✗
AES-GCM	✓	✓	✗
AES-CBC-CMAC	✓	✓	✗
AES Key Wrap	✓	✓	✗
X.509 Zertifikat	✓	✓	✓
TLS 1.2	✓	✓	✗

Tabelle 3-1 Übersicht der Implementierung der BSI-Anforderungen in OpenSSL und OpenSSL.NET

4 Weiterentwicklung von OpenSSL.NET

Um die Entwicklung von OpenSSL.NET voranzutreiben, wird als erstes die Abhängigkeit vom .NET-Framework von der Version 2.0 auf Version 4.5 gesetzt. Für diese Änderung müssen die Projekteigenschaften aller drei Projekte geändert werden. Wie in Abbildung 4.1 ersichtlich ist, wird Reiter „Anwendung“ die Option zur Auswahl des Zielframeworks gefunden. Dort wird das „.NET Framework 4.5“ ausgewählt. Eine weitere optische Änderung, die vorgenommen wird, ist die Umbenennung der Projekte. Zum aktuellen Zeitpunkt haben sie den Suffix „-2010“, welcher suggeriert, dass das Projekt für Visual Studio 2010 gedacht ist. Seit dem Servicepack 1 von Visual Studio 2010 gibt es weitgehend keine Inkompatibilitäten

zwischen den Projektmappen unterschiedlicher Visual Studio Versionen, daher ist der Suffix „-2010“ obsolet und wird aus allen Projektnamen entfernt **/MSDNVS11/** . Für die Verwendung des .NET Framework 4.5 muss allerdings Visual Studio 2012 oder neuer eingesetzt werden. Da zum Zeitpunkt dieser Arbeit bereits an Visual Studio 2013 gearbeitet wird, gibt es einen weiteren Grund auf einen Suffix wie „-2012“ zu verzichten. In der Abbildung 4.1 ist auch das .NET Framework 4.5.1 aufgelistet. Jenes befindet sich zum aktuellen Zeitpunkt jedoch noch in der Testphase. Aus diesem Grund wird von der Verwendung abgesehen. Die Umstellung kann jedoch erfolgen, sobald die finale Version des Frameworks freigegeben ist.

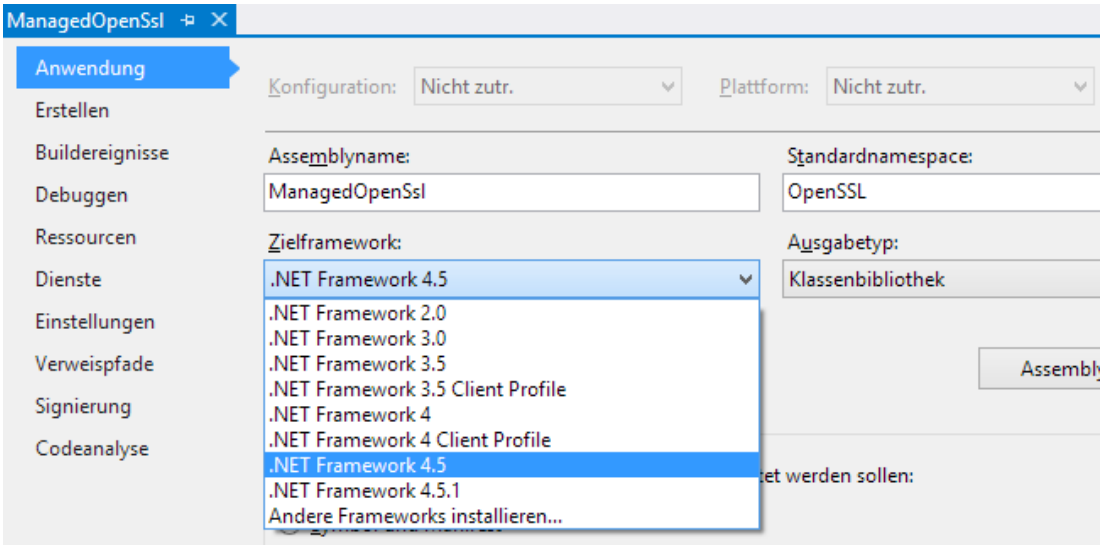

Abbildung 4.1 Auswahl des Zielframeworks

Für einen ersten Test wird in den Projekteinstellungen außerdem die Warnung CS1591, welche besagt, dass ein XML-Kommentar für einen öffentlichen Typen oder Member fehlt, unterdrückt, sodass andere Fehler und Warnungen leichter zu erkennen sind. Erwartet werden Meldungen, dass bestimmte Code-Teile obsolet sind und nicht mehr verwendet werden sollten. Nach dem erfolgreichen Kompilieren bleibt die Fehlerliste leer, das heißt, dass OpenSSL.NET mit der .NET Framework-Version 4.5 kompatibel ist und alter Code somit nicht durch neueren ersetzt werden muss, damit die Bibliothek lauffähig bleibt.

4.1 Umstellung der OpenSSL Version

Im nächsten Schritt wird eine Umstellung der OpenSSL von der Version 1.0.0d auf die aktuelle Version 1.0.1e. Dazu werden zunächst die beiden DLLs im Projekt-Output-Ordner durch die jeweils kompilierte aktuelle Version ersetzt. Das Ausführen eines Tests oder des Kommandozeilenprogramms schlägt, wie in der Abbildung 4.2 zu sehen ist, mit der neuen Version mit einer Ausnahme „Der Typeninitialisierer für "OpenSSL.Core.Native" hat eine Ausnahme verursacht.“ fehl.

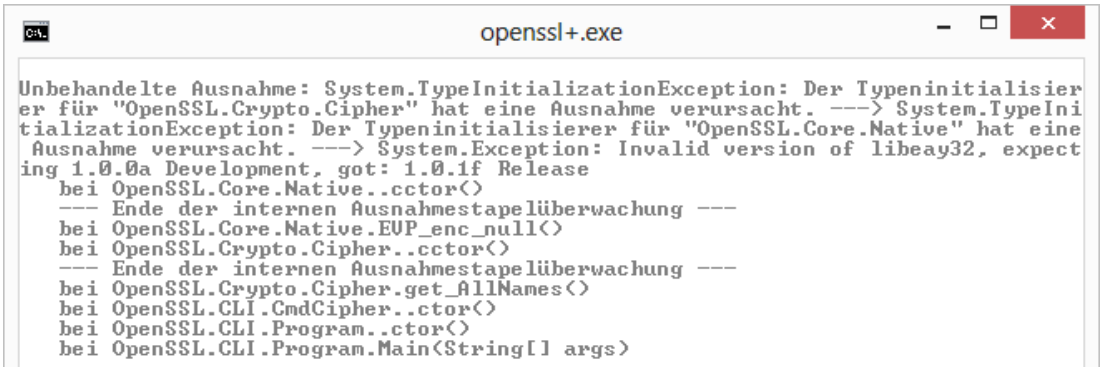

Abbildung 4.2 Fehlerausgabe wegen falscher OpenSSL Version

Im *Debug-Modus* können, wenn die Ausnahme ausgelöst wird, mittels der *IntelliTrace* Funktion von Visual Studio die zuletzt aufgetretenen Ereignisse begutachtet werden. Wichtig sind im diesem Fall die Ereignisse „Debugger“ und „Ausnahme“. Die *IntelliTrace-Liste* wird von oben nach unten gefüllt, das heißt es wird zunächst nach dem Debugger-Ereignis gesucht, welcher einen Breakpoint ausführt, sobald die Ausnahme ausgelöst wird. Vor diesem Debugger-Ereignis befinden sich zwei Ausnahmen. Die untere führt zu der Codestelle, an der das Programm angehalten wurde, dort wurde versucht eine native OpenSSL Funktion aufzurufen. Die obere führt zu der Codezeile, die ausgeführt wäre, wenn die Ausnahme nicht ausgelöst worden wäre. Der Code befindet sich in der `Native`-Klasse und gehört zum statischen Konstruktor dieser Klasse, welcher OpenSSL initialisiert, das heißt Strings werden geladen, die SSL Ciphern und Algorithmen werden registriert und der Zufallszahlengenerator bekommt einen Startwert, der als *Seed* bezeichnet wird. Außerdem wird vor der Initialisierung geprüft, ob die passende OpenSSL Version für den Wrapper benutzt wird, stimmt die erwartete Version nicht mit nativen OpenSSL Version überein, wird die gesuchte Ausnahme ausgelöst.

Für die Verwaltung der Versionen gibt es eine eigene Klasse „`Version`“. Diese Klasse parst[8] ein *Integerwert*[9], der die Struktur der internen OpenSSL Version darstellt in ein verständliches Format. So kann abgefragt werden, ob es eine Entwicklerversion oder die finale Version, und welche Hauptnummer, Unternummer, Fix und Patch es ist. Diese Unterteilung wird mittels Bitweisen-UND und Bitverschiebungen realisiert. In Anbetracht dessen, dass es eine funktionelle `Version`-Klasse gibt, ist es verwunderlich, dass im `Native`-Konstruktor mit den Rohwerten aus `lib.Raw` gearbeitet wird:

```
Version lib = Version.Library;
Version wrapper = Version.Wrapper;
uint mmf = lib.Raw & 0xfffff000;
if (mmf != wrapper.Raw)
    throw new Exception(
        string.Format("Invalid version of {0}, expecting {1}, got: {2}",
            DLLNAME, wrapper, lib));
```

Die Version vom Wrapper ist in der `Native`-Klasse hinterlegt:

```
public const uint Wrapper = 0x10000000;
```

[8] Das Analysieren wird in der Informatik als Parsen bezeichnet.
[9] Ein Integer ist ein ganzzahliger Wert.

Die Version ist ebenfalls in Form der internen OpenSSL Version. Es wird eine Hexadezimale Darstellung verwendet, wobei die einzelnen Ziffern das Format „MNNFFPPS“ annehmen:

- „M“ steht für die Hauptversion (Major),
- „NN“ steht für die Unterversion (Minor),
- „FF“ steht für den Fix,
- „PP“ zeigt den Patch an, und
- „S“ ist der Status, wie zum Beispiel Entwicklerversion, Betaversion 1 bis 14 oder finale Version.

0x10000000 steht in diesem Fall also für die Entwicklerversion der Version 1.0.0. Die zu verwendende Version 1.0.1e entspricht demzufolge 0x1000105F. Da mittels Bitweiser-UND Operation die Version nur auf Hauptversion, Unterversion und Fix verglichen wird, ist es die bessere Alternative den Vergleichsoperator in die `Version`-Klasse zu implementieren. Für inhaltliches Vergleichen wird in der Regel die `Equals`-Methode mit einem Parameter benutzt. In dieser Methode wird die aktuelle Instanz der Klasse mit der übergebenen `Version`-Instanz verglichen. Es wird angenommen, dass die gleiche Version benutzt wird, wenn die Hauptversion, die Unterversion und der Fix identisch sind. Der Vergleich wird über die interne OpenSSL Struktur, die mit einer Bitmaske maskiert ist, ausgeführt:

```
public virtual bool Equals(Version version)
{
    return (Raw & 0xfffff000) == (version.Raw & 0xfffff000);
}
```

Beim Analysieren der Klasse ist auffällig, dass das `Patch`-Feld fehlerhaft implementiert ist:

```
public char Patch
{
       get
       {
              uint patch = (this.raw & 0x00000ff0) >> 4;

              byte a = Encoding.ASCII.GetBytes("a")[0];
              uint x = a + patch;
              char ch = Encoding.ASCII.GetString(new byte[] { (byte)x })[0];
              return ch;
       }
}
```

Es wird zunächst das Patch-Byte korrekt extrahiert. Dieses Byte wird mit dem ASCII-Wert des Buchstabens „a“, welcher den ersten Patch darstellt, addiert, um so den Wert für den aktuellen Patch zu erhalten. Hierbei liegt der Fehler, da es zunächst immer eine Version ohne Patch gibt, wodurch das Patch-Byte der Zahl Null entspricht. Wird der ASCII-Wert des Buchstaben „a“ mit Null addiert, so ergibt es wieder den Wert von „a“, wobei eigentlich kein Zeichen angezeigt werden sollte. Im letzten Schritt wird der ASCII-Wert wieder in ein String umgewandelt, wobei davon das erste Zeichen extrahiert und zurückgegeben wird. Abgesehen davon, dass ein falscher Wert ausgeliefert wird, ist die Funktionsweise durch die vielen Konvertierungen unnötig kompliziert.

Die Funktionsweise für das Feld wird korrigiert:

```
public char Patch {
	get {
		uint patch = (_raw & 0x00000ff0) >> 4;
		return patch != 0
			? (char) ('a' + patch - 1)
			: ' ';
	}
}
```

Es wird zunächst wieder das Patch-Byte extrahiert, welcher als Typ `uint` gesichert wird, um einen Cast, der hier notwendig wäre, zu sparen. Dann wird ein Rückgabewert über einen *ternären Operator*[10] ermittelt: Wenn das Patch-Byte Null ist, wird ein Leerzeichen zurückgegeben, andernfalls wird das Byte auf das Zeichen „a“ addiert und dieser Wert wird um eins dekrementiert und als Zeichen zurückgegeben. In diesem Fall kann der Zahlenwert direkt auf „a“ addiert werden. Dies ist möglich, da das Zeichen in einfachen Anführungsstrichen gesetzt ist, wodurch es als Datentyp `char` deklariert wird. Intern ist der `char`-Datentyp ein numerischer 16 Bit Wert, wodurch es keinen Überlauf geben kann. Der Vorteil dieser Funktionsweise ist es, dass zum einem unnötige Variablen wegfallen und zum anderen finden keine unnötigen Konvertierungen statt, die den Aufruf zwangläufig verlangsamen. In einem Performance Test bei der die alte und neue Methode jeweils eine Million Mal in einer Schleife ausgeführt wurden, erwies sich die neue Version als 20-mal schneller gegenüber der alten.[11]

Des Weiteren wird ein zusätzlicher Konstruktor hinzugefügt, der die zukünftige Pflege des OpenSSL.NET Projekts erleichtern soll. Dieser soll die Möglichkeit bieten, eine Instanz der Version-Klasse zu erstellen, indem man die Version in der Art als Parameter übergibt, wie man Sie als Mensch normal liest. Demzufolge gibt es fünf Parameter, die jeweils einen Teil der Version repräsentieren, diese Werten mittels Bitverschiebungen, Bitweisen-UND und Bitweisen-ODER Funktionen in der Art manipuliert, dass die interne OpenSSL Struktur herauskommt:

```
public Version(byte major, byte minor, byte fix, char patch = ' ',
StatusType status = StatusType.Release)
{
	_raw = (uint) (((major << 28) & 0xf0000000) |
			((minor << 20) & (uint) 0x0ff00000) |
			((fix << 12) & (uint) 0x000ff000));
	if (patch != ' ')
	{
	    _raw |= ((uint) ((patch + 1 - 'a') << 4) & 0x00000ff0);
	}
	_raw |= ((byte) status & (uint) 0x0000000f);
}
```

[10] Ein ternärer Operator, oder auch bedingter Operator, bietet die Möglichkeit in einer Zeile eine Fallunterscheidung durchzuführen und den Rückgabewert je nach Ergebnis variieren zu lassen. Die Syntax baut sich wie folgt auf: `Bedingung ? RückgabewertWahr : RückgabewertFalsch`

[11] Messergebnisse über 1.000.000 Iterationen mittels der `Stopwatch`-Klasse:
Dauer alte Methode: 00:00:00.1833201
Dauer neue Methode: 00:00:00.0096406

Diese Klasse könnte noch mittels Vorinitialisierung oder Verzögerte Initialisierung optimiert werden, was den mehrfachen Aufruf einzelner Felder beschleunigen würde, da die mathematische Operation nur einmal ausgeführt wird. Da die `Version`-Klasse aber zurzeit ausschließlich für den Vergleich beim ersten Aufruf der `Native`-Klasse benutzt wird, ist das Aufwand-Nutzen-Verhältnis zu hoch. In der `Native`-Klasse wird die Wrapper-Versions-Variable durch die neue, besser ersichtliche Form mit der aktuellen Version ersetzt:

```
public static readonly Version Wrapper = new Version(1, 0, 1, 'e');
```

Das letzte, was noch angepasst werden muss, ist der Versionsvergleich im statischen `Native`-Konstruktor. Dabei werden die Bit-Operation und der Vergleich durch die `Equals`-Methode der `Native`-Klasse ersetzt:

```
Version libVersion = Version.Library;
Version wrapperVersion = Version.Wrapper;
if (!libVersion.Equals(wrapperVersion))
    throw new Exception(
        string.Format("Invalid version of {0}, expecting {1}, got: {2}",
            DLLNAME, wrapperVersion, libVersion));
```

Dadurch ist der Code zum einen sauberer, da die Vordergrundlogik, welcher der Versionscheck ist, von der Hintergrundlogik, welche die Bitoperation darstellt, getrennt ist und auch eine weitere Variable eingespart wird. Nach diesen Anpassungen lässt sich OpenSSL.NET auch mit der aktuellen Version von OpenSSL ausführen.

4.2 Entfernen von MD2

Das Kommandozeilentool von OpenSSL.NET bietet wie auch das Original die Möglichkeit Informationen zum benutzten OpenSSL Build auszugeben. Dazu gehören unter anderem auch Optionen für verschiedene unterstützte Algorithmen. Die letzte Änderung am Kommandozeilenprojekt war im August 2009, als OpenSSL 0.9.8k die aktuelle Version war. In der Zwischenzeit wurde mit MD2 einer dieser Algorithmen als unsicher und damit veraltet erklärt. Seit OpenSSL 1.0.0. vom März 2010 ist dieser Hash-Algorithmus standardmäßig deaktiviert. **/OSSLCL/ /OSSLEVPDI/** Dies wiederum führt dazu, dass das Ausführen des Kommandozeilentools mit der Ausgabe der Optionen einen Absturz verursacht, da auf die nicht mehr vorhandene Funktion `MD2_options` von OpenSSL zugegriffen werden sollte. Die Absturzmeldung ist in Abbildung 4.3 zu erkennen.

Abbildung 4.3 Fehlerausgabe wegen fehlender MD2-Funktion

Lösung des Problems ist es den Code, der auf die Funktionen zugreifen will zu deaktivieren. Dafür gibt es mehrere Möglichkeiten:

- Es kann eine Programmlogik eingebaut werden, die über boolesche Variablen bestimmt, ob MD2 aktiviert ist und je nachdem den Code ausführt oder eine Ausnahme auslöst. Diese Variante hätte bis auf die saubere Trennung keinerlei Vorteil gegenüber dem unveränderten Code. Das Kommandozeilentool würde nach wie vor Abstürzen, wenn auch mit einer anderen Fehlermeldung.
- Als zweite Möglichkeit kann der zu MD2 gehörige Code komplett gelöscht werden. Dies bietet den Vorteil, dass das Projekt im kompilierten Zustand je nach *Padding* weniger Speicherplatz auf der Festplatte und im RAM einnimmt. Außerdem wird der Code gekürzt und die überflüssigen Methoden, die in Zukunft wahrscheinlich nicht mehr gebraucht werden, stören die Übersicht nicht.
- Eine letzte Möglichkeit ist das deaktivieren des Codes mittels Präprozessordirektiven. Es kann ein Symbol analog dem im OpenSSL verwendeten „OPENSSL_NO_MD2“ genutzt werden. An den betreffenden Stellen wird der Code mittels Fallunterscheidung ausgegrenzt.

```
#if !OPENSSL_NO_MD2
       Console.Write(" {0}", CryptoUtil.MD2_Options);
#endif
```

Mit „#if“ wird abgegefragt, ob die folgende Bedingung, die in der gleichen Zeile steht, wahr ist. Ist dies der Fall, wird der nachfolgende Code kompiliert, andernfalls wird er beim Kompillieren ignoriert. Die Abfrage wird mittels „#endif“ beendet, es gibt die Möglichkeit mit „#else“ und „#elif“ zusätzliche Fallunterscheidungen zu treffen. **/MSDPRPR/**

Für die Definition des Symbols gibt es ebenfalls verschiedene Varianten. Es kann das Symbol für eine einzelne Datei mittels „#define OPENSSL_NO_MD2“ in der ersten Zeile der Datei gültig gemacht werden. Sinnvoller ist es aber, das Symbol, wie in zu sehen Abbildung 4.4 ist, in den Projekteinstellungen direkt zu hinterlegen. Für den unwahrscheinlichen Fall, dass dieses Symbol für eine einzelne Datei deaktiviert werden soll, kann dies über „#undef OPENSSL_NO_MD2“ in der ersten Zeile der Datei geschehen.

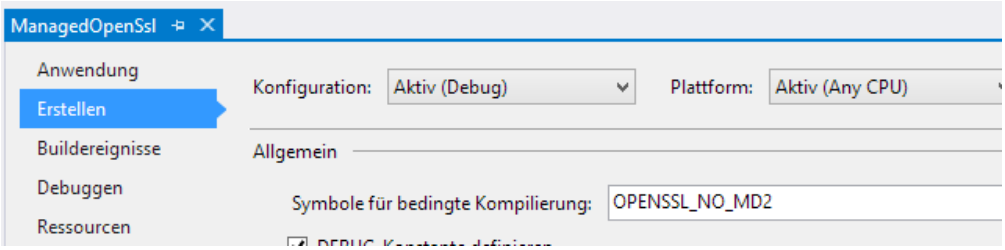

Abbildung 4.4 Definieren der Symbole in den Projekteinstellungen

Die letzte Möglichkeit mit den Präprozessordirektiven wird verwendet, da hiermit die Vorteile der zweiten Möglichkeit inbegriffen sind. Der Quelltext bleibt auch aus dem Grund übersichtlich, da inaktiver Code im Codeeditor einfach ausgeblendet werden kann. Auf diese Weise

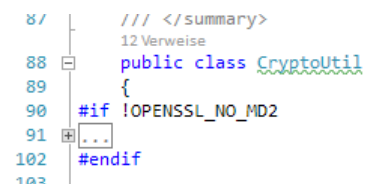

Abbildung 4.5 Ausblenden von inaktiven Code

wurden in Abbildung 4.5 die Zeilen 91 bis 101 ausgeblendet. Zusätzlich gibt es den Vorteil, dass der Code jederzeit wieder aktiviert werden kann, falls die MD2 Funktionalität später noch einmal benötigt wird.

4.3 Kompatibilität der Datentypen

Schon beim Ausführen des Kommandozeilentools mit der Option zur Anzeige der Version fällt auf, dass OpenSSL.NET bei weitem nicht fehlerfrei ist. Das Programm stürzt ab, sobald die OpenSSL Funktion aufgerufen wird, die einen String mit Angaben zur Version zurückliefern soll:

```
[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]
public extern static string SSLeay_version(int type);
```

Das Problem hierbei ist, dass die OpenSSL Bibliothek nicht die Zeichenfolge selbst als Wert zurückgibt, sondern nur einen Zeiger, der auf den String zeigt. Der managed Code versucht mittels Zeiger die Zeichenkette zu ermitteln und gibt anschließend den Speicher frei, um *Memoryleaks* zu vermeiden. Da aber die unmanaged Bibliothek weiterhin auf diesen Pointer zugreifen will, gibt es eine Zugriffverletzung. Dieser Fehler ist wahrscheinlich in der OpenSSL.NET Bibliothek vorhanden, da bisher unter Windows XP entwickelt wurde, welches in diesem Fall keine Zugriffsverletzung auslöst. Stattdessen wurde vom Betriebssystem in Kauf genommen, dass ein möglicher Memoryleak entsteht. Seit Windows Vista wird hierbei die Ausnahme ausgelöst. Um die Probleme aus dem Weg zu gehen, darf der Rückgabewert in C# nicht als Zeichenkette, sondern muss als Pointer gekennzeichnet werden:

```
[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]
public extern static IntPtr SSLeay_version(int type);
```

Weiterhin ist zu beachten, dass nun zusätzlich der Pointer in einen String gemarshallt werden muss, damit dieser auch angezeigt werden kann. In diesem Fall wurde die SSLeay_version-Funktion in der `Version`-Klasse benutzt:

```
public static string GetText(Format format)
{
	return Native.SSLeay_version((int)format);
}
```

Nach dem Ersetzen des Strings durch den Pointer wird die `Marshal`-Klasse benutzt, um einen ANSI-String zu bekommen:

```
public static string GetText(Format format)
{
	IntPtr strptr = Native.SSLeay_version((int)format);
	return Marshal.PtrToStringAnsi(strptr);
}
```

Die Konsolenanwendung wird nun erneut für die Versionsanzeige aufgerufen und es wird, wie gewünscht die Version angezeigt „OpenSSL 1.0.1e 11 Feb 2013“. Die gleiche Verfahrensweise wurde bei den anderen elf Funktionen angewendet, die den Typ `string` zurückgegeben sollen. Wird eine Zeichenkette als Parameter benötigt, muss dieser hingegen nicht als Zeiger deklariert werden, da der Speicher frühestens freigegeben wird, wenn die native Funktion ausgeführt wurde.

Weitere Datentypen die OpenSSL als Parameter akzeptiert sind Integer, Bytearrays, Strukturen und *Callbacks*. Integer und Arrays können als solche direkt übergeben werden. Für Strukturen gibt es zwei Möglichkeiten:

- Die betreffende Struktur wird vor der Übergabe in einen Pointer gemarshallt und dieser Pointer wird vom Typ `IntPtr` als Parameter übergeben. Wird eine Struktur über diese Parameter zurückgegeben, muss der Pointer im managed Code explizit in eine Struktur gemarshallt werden. Beide Möglichkeiten des Marshallens werden hier im Codebeispiel anhand der `ECDSA_SIG_st`-Struktur der `DSASignature`-Klasse gezeigt:

```
private ECDSA_SIG_st Raw {
       get { return (ECDSA_SIG_st)Marshal
              .PtrToStructure(this.ptr, typeof(ECDSA_SIG_st)); }
       set { Marshal.StructureToPtr(value, this.ptr, false); }
}
```

- Die Zweite Möglichkeit für die Übergabe von Strukturen ist, sie direkt als Struktur zu übergeben, sodass das Framework die Umwandlung in einen Pointer übernimmt. Doch damit dies durchführbar ist, muss die Struktur für die `Native`-Klasse sichtbar sein und darf damit nicht als `private` deklariert werden. So müsste sie mindestens als `internal` gekennzeichnet werden. Dies hat den Nachteil, dass alle im OpenSSL.NET *Assembly* vorhandenen Klassen auf die jeweilige Struktur zugreifen können, was der Trennung von den internen nativen OpenSSL-Strukturen und den OpenSSL.NET-Strukturen widersprechen würde. Daher wird hier die die erstere Möglichkeit die zu verwendende sein.

Callbacks sind eine Möglichkeit Funktionen und Methoden als Parameter zu übergeben, die dann zu einem späteren Zeitpunkt ausgeführt werden können. Es wird in C# ein Methodenzeiger erstellt indem zunächst mittels `delegate` Schlüsselwort eine Signatur für den Rückgabetypen und Parameter festgelegt wird. Methoden, die als Callback übergeben werden sollen, müssen die gleiche Signatur aufweisen, wie die dazugehörige *Delegate*. Soll die Methode an unmanaged Code übergeben werden, muss die Delegate mit einen Attribut speziell gekennzeichnet werden.

```
[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
public delegate void MethodHandler(string text, int zahl, IntPtr ptr);

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]
public extern static void NativeMethode(MethodHandler cb);

private static void Handler(string text, int zahl, IntPtr ptr)
{
    //beliebiger Code
}

private static void main()
{
    NativeMethode(Handler);
}
```

In diesem Codebeispiel wird zunächst die Signatur für den die Callbackmethode erstellt, sie heißt hier „`MethodHandler`“ und hat drei Parameter verschiedenen Types. Es wird eine Attribut „`[UnmanagedFunctionPointer(CallingConvention.Cdecl)]`“ angegeben, damit die Delegate benutzt werden kann, um Methodenszeiger an nicht verwalteten Code übergeben zu können. Es wird dadurch die Art und Weise gesteuert, wie .NET den Methodenszeiger marshallt. **/MSDNUFP/** Als nächtes wird für das übergreifende Aufrufen, welches meist kurz als „P/Invoke“ bezeichnet wird, die Signatur der Funktion einer externen nativen DLL deklariert. Als Parameter nimmt diese einen Callback mit der Signatur der vorher definierten Delegaten. Es folgt ein Stub „`Handler`“, welcher der Signatur von „`MethodHandler`“ übereinstimmt. Dies ist die Methode, die aus dem nativen Code heraus aufgerufen werden soll. Im letzten Schritt ist eine `main`-Methode, respektive für jede beliebige Methode, welche die native Funktion ausführt. Es wird der Methodenzeiger von „`Handler`“ übergeben, dies ist nur möglich, da die Signatur der Methode und der Delegaten exakt übereinstimmen. Wurde die nicht verwaltete Funktion aufgerufen, kann die übergebene Methode beliebig oft während der Ausführung der Funktion, zu einem späteren Zeitpunkt, oder gar nicht ausgeführt werden.

4.4 AES

Für die Verwendung in Smart Metering-Systemen werden verschiedene AES-Verfahren als Voraussetzung angesehen. Zu den benötigten Verfahren gehört das *Cipher block chaining* (CBC), der *Galois/Counter Mode* (GCM) und das *Key-Wrap* Verfahren. Die Erstellung von Ciphern mittels CBC wird bereits von OpenSSL.NET unterstützt.

GCM wird von OpenSSL unterstützt, ist aber noch nicht in OpenSSL.NET eingearbeitet. Dies ist aber schnell nachgeholt. Es werden folgende Imports in der Native-Klasse definiert:

```
[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]
public extern static IntPtr EVP_aes_128_gcm();
[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]
public extern static IntPtr EVP_aes_192_gcm();
[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]
public extern static IntPtr EVP_aes_256_gcm();
```

Diese werden in der `Cipher`-Klasse eingebunden. Es gibt für jede Cipher ein statisches Objekt, welches die Funktion aufruft und den zurückgegeben Pointer in der Klasse für eine weitere Verwendung speichert. Für diese Funktionen sieht es wie folgt aus:

```
public class Cipher : Base
{
[…]

       public static Cipher AES_128_GCM
                            = new Cipher(Native.EVP_aes_128_gcm(), false);

       public static Cipher AES_192_GCM
                            = new Cipher(Native.EVP_aes_192_gcm(), false);

       public static Cipher AES_256_GCM
                            = new Cipher(Native.EVP_aes_256_gcm(), false);

[…]
}
```

Die `Cipher`-Klasse wird zum Beispiel verwendet, um den privaten Schlüssel eines Zertifikates zu verschlüsseln und zu speichern.

Für das Key-Wrap Verfahren ist etwas mehr Aufwand notwendig. Auch dieses wird bereits von OpenSSL unterstützt und muss noch in OpenSSL.NET integriert werden. Da noch keine separate Klasse für AES-Funktionen existiert, wird zunächst eine neue Klasse im Namespace `OpenSSL.Crypto` erstellt. Im Gegensatz zu den meisten OpenSSL.NET-Klassen wird kein Kontext für eine einzelne AES-Operation erstellt. Daher wird auf den Verweis auf die `Base`-Klasse verzichtet. Für das Key-Wrap Verfahren werden zunächst zwei Funktionen benötigt, dies sind die `AES_wrap_key` und `AES_unwrap_key` Funktionen. Beide nehmen einen Pointer zu einer AES-Key-Struktur, ein Bytearray für den Initialisierungsvektor, ein Bytearray für den Rückgabewert des zu bearbeitenden Schlüssels, ein Bytearray für den Schlüssel, der übergeben wird, und ein vorzeichenlosen Integerwert für die Länge des Schlüssels als Parameter an. Der Rückgabewert der Funktion ist entweder die Länge des Rückgabebytearrays oder -1 bzw. 0 im Fehlerfall. Beide Funktionen werden der `Native`-Klasse hinzugefügt.

```
[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern int AES_wrap_key(IntPtr key, byte[] iv, byte[] output,
                                      byte[] input, uint inlen);

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern int AES_unwrap_key(IntPtr key, byte[] iv, byte[] output,
                                      byte[] input, uint inlen);
```

Wie eben erwähnt, wird der Pointer einer AES-Key-Struktur benötigt. Diese wird nun in der neuen `AES`-Klasse angelegt. Die Struktur wird aus dem Nativen OpenSSL übernommen und angepasst. Die native Struktur aus OpenSSL sieht wie folgt aus:

```
struct aes_key_st {
#ifdef AES_LONG
    unsigned long rd_key[4 *(AES_MAXNR + 1)];
#else
    unsigned int rd_key[4 *(AES_MAXNR + 1)];
#endif
    int rounds;
};
typedef struct aes_key_st AES_KEY;
```

An C# angepasst ändert sich die Struktur wie folgt:

```
[StructLayout(LayoutKind.Sequential)]
private struct AESKey
{
    private const int AESMaxnr = 14;

    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 4 * (AESMaxnr + 1))]
    private readonly uint[] rd_key;
    private readonly int rounds;
};
```

Wie zu erkennen ist, wurde eine Konstante „`AESMaxnr`“ hinzugefügt. Diese wird ausschließlich in dieser Struktur benutzt. Daher ist es sinnvoll sie innerhalb der Struktur anstatt im Kontext der gesamten Klasse zu definieren. Da die Struktur für die Übergabe – von und zu – nicht verwalteten Code verwendet wird, muss das Array mit einer Größe definiert werden. Dies geschieht mit dem `MarshalAs`-Attribut. Würde dies nicht explizit angegeben

werden, wäre die Größe undefiniert und die Werte wären damit nicht konsistent. Die Umwandlung der Struktur in einen Pointer und umgekehrt, findet wie gewohnt mittels der `Marshal`-Klasse statt. Die Strukturen sind mit Schlüssel für die Ver- und Entschlüsselung befüllt. Das Befüllen wird im Konstruktor der Klasse übernommen, der zwei Parameter annimmt. Der erste Parameter ist ein Bytearray mit dem zu verwendenden Schlüssel und der zweite gibt an wie viel Bit die Verschlüsselung haben soll. Der Konstruktor ruft die beiden privaten Methoden zum Setzen der Schlüssel auf. Bei diesen wird zunächst Speicherplatz für die Struktur allokiert, als Rückgabewert gibt es einen Pointer zu diesem Speicherplatz. Der Pointer wird zusammen mit dem zu verwendenden Schlüssel und die Bitlänge an die native OpenSSL Funktion übergeben. Die Funktion für das Setzen des Schlüssels zum Verschlüsseln heißt „`AES_set_encrypt_key`“. Die Funktion für das Setzen des Schlüssels zum Entschlüsseln heißt „`AES_set_decrypt_key`“. Die Funktionen befüllen die Struktur. Zum Schluss muss nur mittels `Marshal`-Klasse der Pointer in die verwaltete Struktur gewandelt werden. Die Strukturen werden als private Variable in der Klasse gesichert und bei den `WrapKey` bzw. `UnwrapKey`-Methoden wiederverwendet.

Da es zum Kontext der Klasse passt, werden Methoden hinzugefügt, die es ermöglichen Datenblöcke zu ver- und entschlüsseln. Dafür gibt es von OpenSSL die zwei Funktionen „`AES_encrypt`“ und „`AES_decrypt`“, die jeweils ein Bytearray für den Eingabeblock und Ausgabeblock und einen Pointer für den zu verwendenden Schlüssel als Parameter annehmen. Ein Block hat eine Größe von 16 Byte. Werden mehr Daten übergeben, werden diese ignoriert. Ähnlich dem Key-Wrap Verfahren wird zunächst Speicher für die passende Schlüsselstruktur allokiert. Der Rückgabewert der Allokation wird als Pointer zusammen mit den Eingangsdaten und einem initialisierten 16 Byte großen Bytearray an die OpenSSL Funktionen übergeben. Nach der Ausführung der Funktionen sind die verschlüsselten bzw. entschlüsselten Daten in diesen vorher initialisierten Array.

4.5 CMAC

Im Zusammenhang mit AES lohnt es sich, eine weitere Anforderung des BSI bezüglich Smart Metering anzuschauen. Es wird verlangt, dass ein Verfahren mit cipherbasierten Nachrichtenauthentifizierungscode, im konkreten Fall AES-CMAC unterstützt wird. CMAC wurde in OpenSSL.NET bisher nicht implementiert, deshalb muss erneut eine Klasse erstellt werden, welche die im OpenSSL vorhanden Funktionen aufruft. Die Klasse „`CMAC`“ wird analog der „`AES`“ Klasse im Namespace `OpenSSL.Crypto` angelegt. Die CMAC-Funktionen von OpenSSL sind Kontextbasiert, das heißt dass alle Funktionen einen Pointer zur Ausführung benötigen. Aus diesem Grund ist es sinnvoll die `Base`-Klasse als Basisklasse zu benutzen, die die Verwaltung und Freigabe des Kontextpointers übernimmt.

Es wird ein parameterloser Konstruktor erstellt, der nur die Basisklassenkonstruktor aufruft, an den ein Pointer vom zu benutzenden Kontext übergeben wird. In dem Fall, dass der Pointer null ist, wird eine Ausnahme ausgelöst. Wird die Klasse instanziiert, muss dem CMAC-Kontext einen Schlüssel und eine Cipher zugewiesen werden. Dies geschieht mit der OpenSSL Funktion „`CMAC_Init`“. In OpenSSL.NET wird dafür eine Methode „`Init`“ angelegt, die einen Schlüssel als Bytearray und eine Cipher als Parameter annimmt:

```
public void Init(IEnumerable<byte> key, Cipher cipher)
{
    _chipher = cipher;
    byte[] keyBytes = key as byte[] ?? key.ToArray();
    Native.ExpectSuccess(Native.CMAC_Init(ptr, keyBytes,
                    (uint)keyBytes.LongLength, cipher.Handle, IntPtr.Zero));
}
```

Mit der `IEnumerable<byte>`-Schnittstelle wird die Möglichkeit geboten, ein bliebigen Eneumerator, der dem Typ byte auflistet, an die Funktion zu übergeben. So ist sie vielseitiger einsetzbar, da es keine Rolle spielt, ob ein `List<byte>`, ein `byte[]` oder ein anderer Typ, der das `IEnumerable<T>`-Interface implementiert, übergeben wird. Da die native Funktion nur ein Standardbytearray als Parameter annimmt, muss aber erst eine Umwandlung stattfinden. Zunächst wird versucht, den Schlüsselparameter direkt zu als `byte[]` zu casten. Schlägt dies aufgrund von Inkombatiblitäten fehl, würde der Wert „`null`“ zurückgegeben werden. Durch ein doppeltes Fragezeichen „`??`“ kann man diesen `null`-Fall abfangen und stattdessen eine andere Operation ausführen. In diesem Fall wird die `System.Linq`-Erweiterungsmethode „`ToArray`“ benutzt, um das Objekt in ein Bytearray umwandeln. Die zu verwendende Cipher wird in einer privaten Klassenvariable zwischengespeichert. Da von dieser Cipher der Pointer an OpenSSL übergeben werden muss, wird auf das `Handle`-Feld bei der Übergabe zugegriffen.

Ästhetisch besser wäre der Code, wenn das *`Handle`* weggelassen werden könnte. Damit das verwirklicht werden kann, gibt es die Möglichkeit vom .NET Framework Operatoren zu überladen, was wiederum die Freiheit gibt eigene Konvertierungen festzulegen. In diesem Fall ist eine Konvertierung der Klasse in einen Pointer gewünscht. Bei der Überladung von Operatoren gibt es zwei Möglichkeiten. Einerseits kann man implizit konvertieren, was den Vorteil hat, dass nicht einmal ein Cast für die Konvertierung nötig ist, andererseits gibt es explizite Konvertierungen, wobei Casts zwingend erfolderlich sind. Implizite Überladungen sollten nur dann verwendet werden, wenn weder Informationsverlust noch Fehler zu erwarten sind. Da in diesem Fall die Konvertierung nur den Schritt des Aufrufens des Handle-Feldes abnehmen soll, besteht keine der beiden Gefahren, wodurch die implizite Konvertierung benutzt werden kann. Implementiert wird dies in die `Base`-Klasse, damit dieser Vorteil nicht ausschließlich für die `Cipher`-Klasse gültig ist, sondern für alle Klassen, des OpenSSL.NET, die auf die `Base`-Klasse referenzieren.

```
public static implicit operator IntPtr(Base b)
{
    return b.Handle;
}
```

Der obige Codeausschnitt zeigt die fertige Implementierung der Operatorüberladung. Die Überladung muss wie normale Erweiterungen als statisch deklariert werden, als Rückgabewert wird der Pointertyp `IntPtr` angegeben. Die Methode nimmt einen Parameter an, der dem Typ der `Base`-Klasse entspricht. Zurückgegeben wird der Inhalt des `Handle`-Felds. Durch diese Anpassung der Base-Klasse kann das „`cipher.Handle`“ aus dem vorherigen Codeausschnitt durch ein einfaches „`cipher`“ ersetzt werden.

Da die `Init`-Methode vor der ersten Verwendung der CMAC-Funktionen einmal ausgeführt werden muss, kann ein zweiter Konstruktor erstellt werden, der genau diese Aufgabe

übernimmt. Dieser nimmt die gleichen Parameter wie die Init-Methode an. Der Konstruktor führt zunächst die Kontexterstellung aus. Anschließend ruft er die Init-Methode mit beiden Parametern auf.

Für die eigentliche Benutzung des CMACs gibt es im OpenSSL drei Funktionen: CMAC_Update, CMAC_Final und CMAC_resume. Die erste Funktion übernimmt die Aufgabe OpenSSL Daten zu liefern und diese zu verschlüsseln, die zweite schließt den Verschlüsselungsvorgang ab und gibt die verschlüsselten Daten zurück. Die letzte Funktion macht das Abschließen rückgängig, sodass weiterhin Rohdaten zum Verschlüsseln übergeben werden können, bis erneut die CMAC_Final- Funktion aufgerufen wird. Diese drei werden im OpenSSL.NET in serparaten Methoden eingefasst. Die Update-Methode nimmt analog der Init-Methode ein IEnumerable<byte> als Parameter für die zu verschlüsselnden Daten an. Auf die gleiche Weise wird dieser Parameter in die Form eines Bytearrays vom Typ byte[] gebracht. Im nächsten Schritt wird die CMAC_Update-Funktion aufgerufen. Es werden drei Parameter übergeben, dazu gehören der Pointer des CMAC-Kontextes, das Array mit den Daten und die Länge des Arrays. Die Final-Methode soll den Vorgang abschließen und den Nachrichtenauthentifzierungscode zurückgeben. Dafür wird zunächst ein Bytearray mit der Blockgröße der verwendeten Cipher initialisiert. Die Blockgröße kann direkt über das vorher in der Klasse gespeicherten Cipher-Variable aus dem Feld „BlockSize“ ausgelesen werden. Der Zugriff auf das Feld würde eine Ausnahme auslösen, wenn die Init-Methode zuvor nicht ausgeführt wurde. Die Cipher-Variable ist somit kein initialisiertes Objekt und besitzt den Wert „null“. Es muss also zunächst geprüft werden, ob die Variable den null-Wert hat. Sollte dies der Fall sein, wird eine NullReferenceException mit einer passenden Fehlermeldung ausgelöst. Desweiterem muss eine vorzeichenlose Integer-Variable für die Rückgabe der Länge des Nachritenauthentifzierungscodes initialisiert werden. Im Anschluss wird die CMAC_Final-Funktion mit dem Pointer des CMAC-Kontextes, dem initialisierten Array für die Rückgabe des Codes und einer Referenz zu der Integer-Variable als Parameter aufgerufen. Nach der Ausführung ist das Array befüllt und dies wird in der Methode zurückgegeben:

```
public byte[] Final()
{
    if (_chipher == null)
        throw new NullReferenceException(
                        "Cipher is not set, call CMAC.Init() before!");

    var output = new byte[_chipher.BlockSize];
    uint outputlen = 0;
    Native.ExpectSuccess(Native.CMAC_Final(ptr, output, ref outputlen));
    return output;
}
```

Für die CMAC_resume-Funktion wird ebenfalls eine neue Methode „Resume“ erstellt, die lediglich diese Funktion aufruft. Im letzten Schritt ist es wichtig, die OnDispose-Methode der Base-Klasse zu überschreiben. Sie wird aufgerufen, wenn das CMAC-Objekt nicht mehr verwendet werden soll, und mittels Dispose-Methode der nicht verwalteten Speicher – entweder manuell oder per Garbage Collector – freigegeben werden soll. In der OnDispose-Methode wird die CMAC_CTX_free-Funktion aufgerufen, welche den Speicher für den CMAC-Kontext wieder frei gibt.

4.6 CMS

Eine weitere Anforderung ist die Implementierung von *CMS-Funktionen*. CMS steht für *Cryptographic Message Syntax*, eine Syntax die zum Signieren, Verschlüsseln, Entschlüsseln und Verifizieren von Nachrichten benutzt wird. Der aktuelle Standard ist im RFC 5652 beschrieben. **/RFC5652/** Gefordert wurde, dass zumindest das Signieren und Verifizieren funktionstüchtig ist. Es werden diese beiden Funktionen und das Entschlüsseln und Verschlüsseln implementiert werden, was bisher in OpenSSL.NET nicht geschehen ist. Auf die Funktionalitäten zum Komprimieren und Dekomprimieren wird zunächst verzichtet. Diese können später bei Bedarf nachgepflegt werden.

Die CMS Funktionen `CMS_sign`, `CMS_verify`, `CMS_encrypt` und `CMS_decrypt`, sowie die zum Umfeld gehörenden Funktionen benötigten ähnlich der AES-Funktionen (siehe 4.4 AES) keinen Kontext. Diese arbeiten stattdessen mit einer `CMS_ContentInfo`-Struktur, welche einen Inhaltstypen beinhaltet und den dazu gehörigen Inhalt. **/OSSLCMS1/ /OSSLCMS2/ /OSSLCMS3/ /OSSLCMS4/** Die Dokumentation zum CMS-Befehl des Programms für die Kommandozeile lässt darauf schließen, dass keine zwei Operationen nacheinander möglich sind:
„Sign and encrypt mail:

```
openssl cms -sign -in ml.txt -signer my.pem -text \
       | openssl cms -encrypt -out mail.msg \
       -from steve@openssl.org -to someone@somewhere \
       -subject "Signed and Encrypted message" -des3 user.pem“
```

/OSSLCMSCLI/ .
Dieser Befehl führt zuerst die Signierung aus und gibt den erzeugten Text in einen neuen OpenSSL-Prozess über, der diesen Text verschlüsselt. Einige Tests zeigen, dass mehrere Operationen nicht möglich sind. Es kann also nur jeweils eine Ein- und Ausgabe stattfinden, weswegen sich eine statische CMS-Klasse eignet. In diese Klasse werden vier, ebenfalls als statisch deklarierte, Methoden für das Signieren, Verschlüsseln, Entschlüsseln und Verifizieren implementiert.

Für das Signieren von Nachrichten werden neben zwei `BIO`-Objekten für die Ein- und Ausgabe noch zusätzlich Zertifikate mit privaten Schlüsseln benötigt. Die Signatur der Methode sieht wie folgt aus:

```
public static void Sign(BIO input, BIO output,
                        params X509Certificate[] certificateWithPrivateKey)
```

Das `params`-Schlüsselworts vor dem Array mit den Zertifikaten gibt an, dass an diesem Parameter beliebig viele Zertifikate angegeben werden können, ohne diese explizit in einem Array zu kapseln. Diese Funktion kann so benutzt werden, als wäre die Signatur wie folgt deklariert:

```
public static void Sign(BIO input, BIO output,
                        X509Certificate certificateWithPrivateKey1,
                        X509Certificate certificateWithPrivateKey2,
                        X509Certificate certificateWithPrivateKey3,
                        …)
```

Alternativ kann bei der oberen Variante aber auch direkt ein Array übergeben werden. In der Methode selbst kann in beiden Fällen auf alle übergebenen Zertifikaten über das Array `certificateWithPrivateKey` zugegriffen werden.

Für die Signierung muss zunächst die CMS-Struktur erstellt werden, was über CMS_sign-Funktion geschieht. Als Argument werden drei Null-Pointe, die alternativ für ein Signaturzertifikat mit Schlüssel und zusätzlichen Zertifikaten stehen, ein Pointer für das Eingabe-BIO-Objekt und zusätzliche Flags übergeben. Da drei Null-Pointer benutzt werden, können mehrere Zertifikate für die Signatur benutzt werden, dies geschieht im Anschluss über die CMS_add1_signer-Funktion. Diese nimmt den eben erstellten Struktur-Pointer, ein Zertifikat und den dazugehöriger Schlüssel als Parameter. Die Funktion wird in einer Schleife ausgeführt, bis alle Zertifikate benutzt wurden. Bevor der Pointer zur CMS-Struktur wieder freigegeben werden kann, wird mittels SMIME_write_CMS-Funktion das CMS-Objekt in das Ausgabe-BIO-Objekt im *S/MIME-Format* geschrieben. **/OSSLCMS1/**

Für die Verifikation kann das über die SMIME_read_CMS-Funktion erneut eingelesen und in die CMS-Struktur gebracht werden. Anschließend wird die CMS_verify-Funktion aufgerufen, die zum einem die Signatur überprüft und zum anderen den ursprünglichen Text in ein BIO-Objekt schreiben kann. Argumente für diese Funktion ist zum einen der Struktur-Pointer, weitere Zertifikate die benutzt werden sollen, um das richtige, was für die Signatur benutzt wurde zu finden, falls dieses nicht schon in der Struktur vorhanden ist, einen Zertifikatsspeicher, der die nötigen Zertifikate einer Zertifikatskette für die Validierung beinhaltet, einen optionalen Pointer für ein Ausgabe-BIO-Objekt und ein Flag-Parameter. Es wird eine 1 zurückgegeben, falls die Signatur gültig ist, andernfalls wird eine 0 zurückgegeben. Im Anschluss wird in der Methode der Speicher freigegeben und das Ergebnis als boolesche Variable zurückgegeben. **/OSSLCMS2/**

Die Verschlüsselung mittels CMS geschieht über die CMS_encrypt-Funktion, die zum einem einen Stack von Zertifikaten der Empfänger, einen Pointer zu einem BIO-Objekt mit den Eingangsdaten und neben einer Cipher für die Verschlüsselung ein Flag. Zurückgegeben wird ein Pointer zu der CMS-Struktur, der wiederum in der SMIME_write_CMS-Funktion benutzt wird, um die Struktur im S/MIME-Format abzuspeichern. Anschließend wird der Speicher wieder freigegeben. **/OSSLCMS3/**

Für die Entschlüsselung wird zunächst über die SMIME_read_CMS-Funktion eine S/MIME-Datei in die CMS-Struktur geladen. Ab diesem Punkt gibt es zwei Möglichkeiten. Eine wäre es über die CMS_decrypt-Funktion die CMS-Struktur direkt mit den angegebenen Zertifikat und Schlüssel zu entschlüsseln und in ein Ausgabe-BIO-Objekt zu schreiben. Eine Alternative ist es, den Schlüssel und das Zertifikat vorher über die CMS_decrypt_set1_pkey-Funktion zu setzen und anschließend die CMS_decrypt-Funktion mit Null-Pointern anstelle des Zertifikats und des Schlüssels aufzurufen. Der Vorteil der zweiten Methode ist es, dass eine spätere Modifikation, die die Angabe mehrere Empfängerzertifikate ermöglicht, sehr einfach umzusetzen ist, da die CMS_decrypt_set1_pkey-Funktion nur in einer Schleife über alle angegebenen Schlüssel und Zertifikate laufen muss. Die Angabe des Zertifikats ist optional, daher kann auch ein Null-Pointer ausgewählt werden, was zur Folge hat, dass alle möglichen Empfänger durchprobiert werden. **/OSSLCMS4/**

4.7 Brainpool Kurven

Für die Elliptische Kurven Kryptographie (ECC) gibt es verschiedene Standards, die verwendet werden können. Dazu gehören unter anderem Kurven über Primkörpern, die von einer Bundesbehörde der Vereinigten Staaten – dem *National Institute of Standards and Technology* (NIST) – standardisiert wurden. Dazu existieren als Alternative die

Brainpoolkurven, welche vom *ECC-Brainpool*, eine Arbeitsgruppe, die aus verschiedenen Firmen und Institutionen besteht, die sich mit dem Thema Elliptische Kurven Kryptographie beschäftigen, spezifiziert und durch das IETF im RFC 5639 **/RFC5639/** standardisiert wurden **/ECCBP/** . Das BSI fordert, dass für Smart Metering Systeme beide oben genannten Standards unterstützt werden müssen, wobei in den nächsten Jahren die NIST-Kurven durch die Brainpoolkurven abgelöst werden sollen. Im Standard OpenSSL sind nur die NIST-Kurven implementiert, die bereits im OpenSSL.NET integriert sind. Die Brainpoolkurven hingegen sind weder im OpenSSL noch im OpenSSL.NET implementiert. Es existieren aber Patches von der OpenSSL-Community, die diese Kurven nachrüsten **/BPECC/** . Für das OpenSSL.NET Projekt hat es zur Folge, dass nun entweder die Standardbibliothek ohne den Brainpoolkurven unterstützt wird, oder nur die modifizierte Version. Eine dritte Möglichkeit ist es, die Brainpoolkurven im OpenSSL.NET separat zu implementieren, sodass das Projekt weiterhin mit dem Standard OpenSSL kompatibel bleibt. Die Entscheidung fällt darauf, dass beide Versionen unterstützt werden und ähnlich wie bei dem MD2 Verfahren mittels Symbol (siehe Kapitel 4.2) zwischen der in OpenSSL.NET implementierten und der nativen im OpenSSL gepatchten Version umzuschalten.

4.7.1 Unmanaged Lösung

Zunächst wird aufgrund der schnelleren Umsetzbarkeit die native Version implementiert. Dafür wird in der `Objects`-Klasse aus dem `OpenSSL.Core`-Namespace angepasst. In dieser Klasse werden auf die Kurvenfunktionen mittels ASN.1-ID, der NID, oder dem Kurznamen der Kurve verwiesen. In der gepatchten OpenSSL-Version werden die NIDs 921 bis 934 für die Brainpoolkurven verwendet. Es könnten diese NIDs verwendet werden, um auf die richtigen Kurven zu verweisen, wobei das Problem bestehen bleibt, dass diese Kurven nicht Standardmäßig in OpenSSL implementiert sind, und daher diese IDs je nach Implementierung abweichen können. Die bessere Alternative stellt hierbei der Verweis über den Kurznamen dar. Für die ASN.1-Objekte, die über die Namen indiziert werden, gibt es in der `Objects`-Klasse eine Unterklasse „`SN`“. In dieser Unterklasse werden die Brainpool-Objekte angelegt. Am Beispiel der Brainpool P160r1 Kurve wird „brainpoolP160r1“ als Kurznamen übergeben:

```
public static Asn1Object brainpoolP160r1 = new Asn1Object("brainpoolP160r1");
```

Sowohl die ID, als auch der Kurzname sind in der gepachten Version in der „obj_mac.c“ Datei zu finden **/BPOBJ/** . Die ASN.1-Objekte können dafür verwendet werden, um Schlüssel für Elliptische Kurven zu initialisieren, was über die `FromCurveName`-Methode der `OpenSSL.Crypto.EC.Key`-Klasse geschieht.

Um die Möglichkeit, zwischen der OpenSSL und OpenSSL.NET Version der Brainpoolimplementierung wechseln zu können, zu realisieren, wird eine separate `Brainpool`-Klasse im `OpenSSL.Crypto.EC`-Namespace erstellt. In dieser Klasse wird zunächst eine Enumeration angelegt, die jeweils einen Eintrag für jede Kurve enthält. Über diese Enumeration wird später ausgewählt, mit welcher Kurve ein Schlüssel initialisiert werden soll. Desweiteren wird eine Methode „`Create`“ erstellt, die einen Wert der Enumeration als Parameter annimmt. Es wird eine Fallunterscheidung ausgeführt und je nach übergebenen Parameter wird der passende Schlüssel mit der bereits erwähnten `FromCurveName`-Methode erstellt. Sollte kein passender Eintrag gefunden werden, wird eine Ausnahme ausgelöst:

```
public static Key Create(Brainpools brainpool)
{
    switch (brainpool)
    {
        case Brainpools.P160r1:
            return Key.FromCurveName(Objects.SN.brainpoolP160r1);
[…]
        case Brainpools.P512t1:
            return Key.FromCurveName(Objects.SN.brainpoolP512t1);
        default:
            throw new ArgumentOutOfRangeException("brainpool");
    }
}
```

Die Implementierung der nativen Methode ist damit abgeschlossen. Im nächsten Schritt wird die Alternativlösung im OpenSSL.NET ausgearbeitet. Für diesen Zweck wird die `Brainpool`-Klasse erweitert. Zunächst wird aber die Definition der Brainpool-ASN.1-Objekte in der `Objects`-Klasse mittels Präprozessordirektive „`#if !NATIVE_BRAINPOOL`“ deaktiviert. Selbiges geschieht mit den obigen `case`-Anweisungen.

4.7.2 Managed Lösung

Um Elliptische Kurven zu erstellen, werden sechs Angaben benötigt. Hier werden diese als „Prime“, „A“, „B“, „Gx“, „Gy“ und „Order“ bezeichnet. Diese bilden zusammen als Bytearrays innerhalb einer neu angelegten Struktur die Grundlage für die Definition der Kurvenparameter:

```
private struct GeneratorStruct
{
    public byte[] A, B, Order, Gx, Gy, Prime;
}
```

Es werden nun Objekte erstellt, die diese Strukturen mit den korrekten Parameter, die sowohl dem RFC 5639 **/RFC5639/** als auch einem offiziellem Dokument der ECC Brainpoolgruppe **/BPDP05/** entnommen werden können. Auszugsweise ist das Vorgehen in einem Beispiel zur Brainpoolkurve P160r1 zu erkennen:

```
private static readonly GeneratorStruct P160r1 = new GeneratorStruct
{
    A = new byte[]
    {
        0x34, 0x0E, 0x7B, 0xE2, 0xA2, 0x80, 0xEB, 0x74, 0xE2, 0xBE, 0x61,
        0xBA, 0xDA, 0x74, 0x5D, 0x97, 0xE8, 0xF7, 0xC3, 0x00
    },
[…]
    Prime = new byte[]
    {
        0xE9, 0x5E, 0x4A, 0x5F, 0x73, 0x70, 0x59, 0xDC, 0x60, 0xDF, 0xC7,
        0xAD, 0x95, 0xB3, 0xD8, 0x13, 0x95, 0x15, 0x62, 0x0F
    }
};
```

Nachdem alle zwölf Kurven parametrisiert wurden, kann die eigentliche Implementierung beginnen. Dazu wird die `ec_group_new_from_data`-Funktion aus OpenSSL **/BPECC/** entsprechend nachgebaut und angepasst. Die `CreateKey`-Methode nimmt eine von den oben angelegten Strukturen als Parameter an und gibt eine funktionstüchtige `Key`-Klasseninstanz zurück. Als erstes müssen diese Arrays in `BigNumber`-Objekte gewandelt

werden. Zu diesen sechs Objekten muss noch ein zusätzlicher BigNumber-Kontext erstellt werden, der für die interne Verarbeitung beim Erstellen der Kurven benötigt wird. Es werden nacheinander vier verschiedene OpenSSL-Funktionen aufgerufen:

1. Die Funktion EC_GROUP_new_curve_GFp erstellt mithilfe der Parameter Prime, A und B eine Gruppe für Elliptische Kurven erstellt, die der Form $y^2 = x^3 + a * x + b$ entsprechen.
2. Danach wird mittels des zurückgegeben Pointers die Funktion EC_POINT_new aufgerufen. Es wird ein Pointer zu für einen zu definierenden Punkt auf der Kurve zurückgegeben.
3. Es wird mit dem Gruppenpointer und dem Punktpointer eine weitere affine Koordinate auf der Kurve festgelegt. Der Punkt wird über die Parameter Gx und Gy bestimmt, die an die EC_POINT_set_affine_coordinates_GFp-Funktion übergeben werden.
4. Im letzten Schritt werden mittels der Funktion EC_GROUP_set_generator die Kurvenparamter festgelegt. Diese sind der Generator der Kurve, der im dritten Schritt durch den für kryptographische Operationen wohldefinierten Punkt erstellt wurde, der Ordnung der Kurve und ein Cofaktor. Die Ordnung multipliziert mit dem Cofaktor ergeben die Anzahl der möglichen Punkte auf der Kurve. Die Ordnung wird durch den Order-Parameter und der Cofaktor durch den Gx-Parameter bestimmt.

```
//create the curve
IntPtr groupPtr = Native.EC_GROUP_new_curve_GFp(prime, a, b, ctx);
//create the generator
IntPtr pointPtr = Native.EC_POINT_new(groupPtr);
//set coordinates for generator
Native.EC_POINT_set_affine_coordinates_GFp(groupPtr, pointPtr, x, y, ctx);
//set the generator and the order
Native.EC_GROUP_set_generator(groupPtr, pointPtr, order, x);
```

Die Funktionen aus Punkt 1 und 3 benötigen zusätzlich den vorher erstellten BigNumber-Kontext-Pointer als Parameter. Durch diese Schritte wurde eine Gruppe für Elliptische Kurven erzeugt. Im folgenden wird der Speicher für den erstellten Punkt aus Schritt 2, die sechs BigNumber-Objekte und der zusätzlich erstellte BigNumber-Kontext wieder freigegeben. Es wird der Gruppenpointer aus dem ersten Schritt benutzt, um eine Group-Klasse zu instanzieren. Die Instanz wird einer neu erstellten Key-Instanz zugeweisen, wodurch eine fertiges Key-Objekt erzeugt wird, was von der CreateKey-Methode zurückgegeben werden kann.

Um die Kompatiblität mit dem Standard OpenSSL zu wahren, wird darauf verzichtet, den Kurven einen einen Namen bzw. einer OID zuzuweisen. Ohne diese Zuweisung werden beispielsweise den mit diesen Kurven erstellten Zertifikaten die Parameter explizit mit hinzugefügt. So kann OpenSSL anhand dieser Daten die Kurve generieren und das Zertifikat validieren. Wird der Kurve allerdings ein Name zugewiesen, erscheint im Zertifikat lediglich diese kurze ID, die bei allen zu verwendenden Anwendung einheitheitlich sein muss. Da die Erweiterung der Brainpoolkurven vom Standard abweicht, wäre die Einhaltung der Einheitlichkeit nicht gewährleistet.

In der vorher angelegten Create-Methode werden erneut case-Anweisungen für die Brainpools-Enumeration eingefügt. Dieses Mal wird die eben erstellte CreateKey-Methode aufgerufen. Der Paramter für diese ist jeweils eine der definierten Kurvenparameter.

Um die Abgrenzung zu der nativen Implementierung sauber abzurunden, wird die gesamte .NET Implementierung in die Präprozessordirekte mit eingeschlossen, sodass weder die Kurvenparameter, die Struktur für diese, noch die CreateKey-Methode kompiliert wird, wenn dies nicht nötigt ist. Wird das Projekt mit dem „NATIVE_BRAINPOOL“ Symbol kompilliert, enthält die `Brainpool`-Klasse nur die `Create`-Methode und die Enumeration mit allen Brainpoolkurven. Ohne dem Symbol wird die Klasse im gesamtem Funktionsumfang – mit Ausnahme der `case`-Teile, die auf die ASN.1-Objekte zeigen würde – kompilliert.

4.8 TLS und SSL

Das wichtigste für das Smart Metering Projekt ist es, dass eine Netzwerkkommunikation aufgebaut werden kann. Wie sich in der Analyse des OpenSSL.NET Projektes zeigte, ist diese zum aktuellen Zeitpunkt nur mangelhaft implementiert. Für die SSL-Funktionalität gibt es acht Klassen:

- Ssl,
- SslCipher,
- SslContext,
- SslMethod,
- SslStream,
- SslStreamBase,
- SslStreamClient und
- SslStreamServer.

Die Zusammenhänge der Klassen wurde bereits im Kapitel 2.1 erläutert und in der Abbildung 2.1 zusammengefasst. Zusätzlich gibt es eine weitere Datei „Enums.cs“, welche verschiedene Enumerationen für die SSL-Funktionen beinhaltet. Im folgendem werden die wichtigen Klassen einzeln beschrieben und Korrekturen vorgenommen. Als erstes wird die Funktionstüchtigkeit des Clients sichergestellt. Generell werden in .NET Clientverbindungen aufgebaut, indem zunächst durch die `TcpClient`-Klasse eine Netzwerkverbindung hergestellt wird. Steht die Verbindung, kann mithilfe `GetStream`-Methode der Stream, welcher für das Senden und Empfangen von Daten benutzt wird, zurückgegeben werden. Werden auf dieses `Stream`-Objekt Daten geschrieben, kann eine unverschlüsselte Verbindung Daten versenden. Allerdings kann eine `SslStream`-Instanz des .NET Frameworks mit dem Stream als Parameter erstellt werden, um eine verschlüsselte Verbindung mit dem SSL Protokoll aufzubauen. Wie vom BSI für Smart Metering verlangt, wird auch TLS 1.2 unterstützt. Allerdings sind beispielsweise Zertifikatauthentifizierungen von Zertifikaten, die mit einem Brainpoolschlüssel versehen sind, nicht möglich. Um dieses Defizit zu beseitigen, wird die SSL/TLS-Funktionalität von OpenSSL benutzt. Für diesen Zweck gibt es eine `SslStream`-Klasse, die anlehnend an die original Framework-Klasse in OpenSSL.NET implementiert wurde.

Sie implementiert, wie die Standardklasse auch, die abstrakte `AuthenticatedStream`-Klasse, welche die Grundlage für authentifizierte Streams darstellt. Durch das Benutzen von abstrakten Klassen entsteht eine Interoperabilität, was den Austausch der Klasse für den gesicherten Netzwerkverkehr vereinfacht, da diese die gleichen Methoden implementieren, die zum Senden und Empfangen von Daten benötigt werden. In der OpenSSL.NET-`SslStream`-Klasse gibt es zwei besonders wichtige Methoden, welche für die Authentifizierung vom Client bzw. Server zuständig sind: die `AuthenticateAsClient` und die `AuthenticateAsServer` Methoden.

4.8.1 Client

Die `AuthenticateAsClient`-Methode wird direkt nach dem Initialisieren des `SslStream`s aufgerufen. Es gibt zwei Überladungen[12]. Eine Methode nimmt nur den Namen des Zielservers an und erstellt so eine verschlüsselte Verbindung mit den Standardeinstellungen, wobei nur die Authentizität des Servers geprüft wird. Die zweite nimmt ebenfalls den Hostnamen an, zusätzlich aber auch ein Clientzertifikat für die Clientauthentifizierung, eine Zertifikatkette, die alle Zwischenzertifikate vom Clientzertifikat bis zum Root-Zertifikat beinhaltet, um das Clientzertifikat validieren zu können. Weiterhin werden die zu erlaubten SSL-Protokolle und die Stärke der zu verwendenden Cipher Suite mit angegeben. Ein letzter Parameter ist die Angabe, ob eine Zertifikatsperrliste verwendet werden soll. Diese Funktion wurde in OpenSSL.NET aber noch nicht implementiert. Aus dieser Methode heraus wird ein `SslStreamClient` erstellt, der die Funktionen für einen Client beinhaltet.

In der Initialisierung der `SslStreamClient`-Klasse wird zunächst ein Kontext mit der `SSLv23_client_method`-Funktion erstellt. Diese Funktion erlaubt laut OpenSSL-Dokumentation SSLv2, SSLv3 und TLS 1.0 Verbindungen **/OSSLCTX/** . Da laut BSI für die Netzwerkkommunikation aber mindestens TLS 1.2 benötigt wird, muss der Kontext ersetzt werden. Es könnte direkt die `SSLv23_client_method`-Funktion durch `TLSv1_2_client_method` ersetzt werden. Dies beherbergt aber wieder die Nachteile der festen Programmierung, dass die Klasse danach nicht mehr flexibel für ältere Protokolle einsetzbar sein würde. Da die erlaubten SSL-Protokolle aber an die Methode mit übergeben werden, kann anhand einer einfachen Fallunterscheidung bestimmt werden, welche Kontext-Funktion aufgerufen werden soll. TLS 1.2 wurde in OpenSSL.NET noch nicht integriert. Aus diesem Grund wird dies nachgeholt. In der vorher erwähnten Enums-Datei befinden sich verschiedene Enumerationen zu verschiedenen Algorithmen und Protokollen, wozu auch die Protokolle für SSL und TLS gehören. Wie rechts im Codeausschnitt erkennbar ist, sind alle Protokolle von SSL 2 bis TLS 1.0 vorhanden, TLS 1.1 und TLS 1.2 fehlen. Über jedem Eintrag befindet sich ein XML-Kommentar, der für die IntelliSense-Funktion von Visual Studio und auch für eine automatisch generierte Dokumentation verwendet wird. Die Einträge sind alle leer. Für das bessere Verständnis bei der Entwicklung werden diese Kommentare entsprechend befüllt, indem das dazugehörige Protokoll an die entsprechende Stelle eingetragen wird. Bei `Tls` wird beispielsweise „TLSv1“ eingefügt und bei `None` erscheint ein Hinweis, dass dieser Eintrag nur für uninitialisierte Werte benutzt werden sollte. Es werden die Einträge für die fehlenden TLS-Versionen ergänzt. TLS 1.1 erhält den Zahlenwert 8 und TLS 1.2 den Wert 16. Es entsteht ein Konflikt zum `Default`-Wert, daher wird dieser entsprechend seiner Bedeutung angepasst. Mittels ODER-Verknüpfung bekommt er den

```
public enum SslProtocols
{
        /// <summary>
        ///
        /// </summary>
        None = 0,
        /// <summary>
        ///
        /// </summary>
        Ssl2 = 1,
        /// <summary>
        ///
        /// </summary>
        Ssl3 = 2,
        /// <summary>
        ///
        /// </summary>
        Tls = 4,
        /// <summary>
        ///
        /// </summary>
        Default = 16
}
```

[12] Eine Überladung ist das Implementieren einer gleichnamigen Methode, die andere Parameter annimmt und gegeben falls auch einen anderen Rückgabetypen hat.

Wert von `Ssl2`, `Ssl3` und `Tls`. Wenn wie in diesem Fall eine Bit-Weise Verknpüfung benutzt wird, wird empfohlen die Enumeration mittels `Flags`-Attribut zu kennzeichnen **/MSDNFLG/** . Am Ende sieht die Enumeration wie folgt aus:

```
[Flags]
public enum SslProtocols
{
    /// <summary>
    /// Standard uninitialised value, do not use it
    /// </summary>
    None = 0,
[…]
    Tls1_1 = 1 << 3,
    /// <summary>
    /// TLSv1.2
    /// </summary>
    Tls1_2 = 1 << 4,
    /// <summary>
    /// SSLv2, SSLv3, and TLSv1
    /// </summary>
    Default = Ssl2 | Ssl3 | Tls
}
```

Nach dieser Ergänzung müssen außerdem die *TLS-method*-Funktion der `SslMethod`-Klasse hinzugefügt werden. Zu beiden Protokollen gibt es drei Funktionen, die sich durch ihr Suffix unterscheiden. Die „method“-Funktion erstellt ein Kontext, der nicht nur für den Client sondern auch für den Server allgemeingültig ist. Die „server_method“ und „client_method“ schränken die Anwendung jeweils ausschließlich auf den Server oder den Client ein. **/OSSLCTX/** Nachdem diese sechs Funktionen sowohl in der `Native`-Klasse als auch in dieser `SslMethod`-Klasse zu den bereits vorhandenen ergänzt wurden, kann weiter an der Initialisierung des `SslStreamClient`s gearbeitet werden.

Es fand die Auswahl des korrekten SSL-Protokolls statt. Wird kein passendes Protokoll gefunden, löst dies eine Ausnahme aus, welche beschreibt, dass das ausgewählte Protokoll nicht unterstützt wird. Im nächsten Schritt wird überprüft, welche Protokolle nicht übergeben wurden. Diese werden über die SSL-Kontext-Optionen deaktiviert. Die Vergleiche werden über bitweise-Operationen ausgeführt.

Um die Übersicht im Code zu verbessern, gibt es seit dem .NET Framework 4 für Enumerationen die Methode „`HasFlag`“, womit auf einfachste Weise geprüft werden kann, ob ein bestimmter Wert gesetzt wurde. Nachteil dieser Methode ist es, dass diese nicht generisch ist, was zur Folge hat, dass sowohl die zu überprüfende Enumeration als auch der Wert, der geprüft werden soll, vorübergehend in eine `Enum`-Klasse *geboxt* werden muss. Die Umwandlung des Wertetyps in einen Referenztyp (*boxing*) kostet Zeit, sodass die Methode viel langsamer wird als der bitweise-Vergleich. Sollte dieses Problem mit einer späteren .NET Framework-Version behoben werden, können die bitweisen Operationen ersetzt werden. Zum aktuellen Zeitpunkt wird jedoch auf diese Methode zugunsten der Performance verzichtet. **/MSD13/ /MSDBOX/**

Dadurch, dass die Enumeration zu den SSL-Protokollen überarbeitet wurde, kann die vorhandene Abfrage nach dem Default-Wert entfernt werden. Die Prüfung des `Default`-Werts wird implizit durch die Überprüfung der dazugehörigen Protokolle SSLv2, SSLv3 und TLSv1 ausgeführt.

Weiterhin wird eine Liste von Unterstützen *Cipher Suites* festgelegt. Diese wird dynamisch anhand der übergebenen Protokolle und der Cipher-Stärke ermittelt. Das Zusammenfügen der Teile dieser Liste geschieht über langsame `String`-Operationen. Diese werden durch die vielfach schnellere Alternative dem `StringBuilder` ersetzt. Obwohl es maximal nur sechs Operationen sind, bringt die Ersetzung eine Zeitersparnis von ein Viertel der ursprünglich benötigten Zeit.[13] Zusätzlich wurden die verschachtelten Fallunterscheidungen durch ternäre Operatoren ersetzt, sodass der Code besser lesbar ist. Das BSI fordert, dass nur bestimmte *Cipher Suites* unterstützt werden dürfen. In diesem Fall darf die Liste nicht dynamisch aufgebaut werden, stattdessen muss sie explizit festgelegt werden. Die Liste kann nur während der Kontextinitialisierung, also noch bevor SSL selbst mit diesem Kontext initialisiert wird, gesetzt werden. Aus diesem Grund wird allen Methoden in der Aufrufhierarchie ein optionaler Parameter, der diese Liste als Zeichenkette beinhalten kann, hinzugefügt. Ist der Parameter befüllt, wird dieser zum Setzen der Cipher-Liste benutzt, andernfalls wird sie dynamisch erzeugt.

Im den nächsten Schritten wird ein Callback für die Auswahl des Clientzertifikats, gegebenenfalls ein Callback für die Validierung des Serverzertifikats und ein Zertifikatspeicher, der für die Validierung des Clientzertifikats nötigen Zertifikate beinhaltet, gesetzt. Diese Funktionen benötigen keine Modifikation. Danach wird der Kontext benutzt, ob ein `Ssl`-Objekt zu erstellen. Dieser Vorgang gehört inhaltlich nicht mit in die `InitializeClientContext`-Methode und wird daher in den Konstruktor der `SslStreamClient`-Klasse ausgelagert.

Eine weitere Forderung vom BSI gibt an, dass eine Session im WAN bzw. HAN nur zwei Tage, im LMN hingegen ein Monat aktiv bleiben darf. Daher muss zusätzlich ein *Timeout* festgelegt werden. Dafür wird in der `SslContext`-Klasse ein Feld „Timeout“ angelegt, welches eine `set` und eine `get`-Methode hat. Das Feld ist vom Typ `TimeSpan`, um Zeitangaben einfach umwandeln zu können. Über die `SSL_CTX_set_timeout`-Funktion wird der Timeout in Sekunden festgelegt. Dies geschieht über das `TotalSeconds`-Feld, welches die im `TimeSpan`-Objekt hinterlegte Zeitspanne in Sekunden umrechnet. Die `SSL_CTX_get_timeout`-Funktion gibt das gesetzte Timeout in Sekunden zurück:

```
public TimeSpan Timeout
{
	set { Native.SSL_CTX_set_timeout(ptr, (long) value.TotalSeconds); }
	get { return TimeSpan.FromSeconds(Native.SSL_CTX_get_timeout(ptr)); }
}
```

Problem bei diesem Feld in der obigen Form ist der Rückgabewert. OpenSSL legt als Standardtimeout eine Zahl fest die weit über das Maximum von `TimeSpan` hinausgeht, woraufhin eine Ausnahme ausgelöst werden würde. Um einen Programmabsturz zu verhindern, wird diese Ausnahme abgefangen und ein `TimeSpan`-Objekt wird mit dem Wert Null zurückgegeben. Dadurch entsteht natürlich ein Fehler, der in diesem Fall bei bis zu 10^7 Sekunden liegen kann. Wird dieser Wert erneut zum Setzen des Timeouts benutzt, wird der Standardwert von OpenSSL verwendet. Dieser Fehlerfall würde ab einem Timeout-Wert von über 29.000 Jahren eintreten. Dies liegt weit über den vorstellbaren Anwendungsfällen.

[13] Messergebnisse über 1.000.000 Iterationen mittels der `Stopwatch`-Klasse:
Dauer alte Methode: 00:00:00.4255558
Dauer neue Methode: 00:00:00.3338945

Der Timeout wird in der Kontextinitialisierungsmethode gesetzt, wodurch der Kontext vollständig beschrieben ist. Als nächstes wird das `Ssl`-Objekt erstellt, welches die gesamte Verbindung verwaltet. Ab diesen Punkt ist die Handhabung zwischen Client und Server weitgehend identisch.

Wichtig zu erwähnen ist die Zertifikatauswahl des Clients. Dies geschieht über den oben erwähnten Callback. Es werden alle CA-Zertifikate, die vom Server übergeben wurden, abgefangen und die Aussteller in eine Liste geschrieben. Wurde ein entsprechender `LocalCertificateSelectionHandler`-Callback beim Erstellen des `SslStream` übergeben, wird dieser aufgerufen, um ein Zertifikat auszuwählen. Das Zertifikat und der dazugehörige private Schlüssel werden über jeweils ein Output-Parameter zurückgegeben. Eine Anpassung des Callbacks findet insofern statt, dass auch ein Zertifikat zurückgegeben wird, wenn kein *Selection-Handler* angegeben wurde. Dabei wird das erste Zertifikat ausgewählt, wo der Aussteller des Zertifikats mit einem Antragssteller der CA-Zertifikate übereinstimmt.

4.8.2 Server

Der Vorgang zum Erstellen eines `Ssl`-Objekts für einen Server mittels `AuthenticateAsServer`-Methode ist dem mittels `AuthenticateAsClient`-Methode sehr ähnlich. Als Parameter wird zwingend ein Serverzertifikat übergeben. Weiterhin kann angegeben werden, ob der Client ein Zertifikat zum Server senden soll. Es kann eine Zertifikatkette für die Validierung des Serverzertifikats, die erlaubten SSL-Protokolle und Cipherstärken mit übergeben werden. Ein weiterer Parameter ist die Angabe, ob ein Zertifikat zusätzlich über eine Zertifikatsperrliste validiert werden soll. Diese Funktion ist bis dato nicht in OpenSSL.NET implementiert. Es wird ein `SslStreamServer`-Objekt erstellt, welches analog der `SslStreamClient`-Klasse den SSL-Kontext erzeugt und daraus ein `Ssl`-Objekt erstellt. Statt den „client_method“-Funktionen werden die „server_method“-Funktionen zum Initialisieren des `SslContext`s verwendet. Ein Spezifikum für das Serverumfeld ist das Festlegen der Verifizierung. Über die `SSL_CTX_set_verify`-Funktion wird festgelegt, ob und wie ein Client-Zertifikat verifiziert werden soll. Zusätzlich wird ein Callback angegeben, der für eine genauere Validierung benutzt werden kann. Mittels `SetVerifyDepth`-Methode der `SslContext`-Klasse wird angegeben, dass bei einer Validierung die CA-Zertifikate bis zu 10 Hierarchiestufen durchlaufen werden kann, bevor ein Fehler ausgegeben wird. Wenn CA-Zertifikate für Validierung mit übergeben wurden, wird zunächst ein Zertifikatspeicher mit den entsprechenden Zertifikaten für den Kontext festgelegt. Anschließend werden diese Zertifikate in Form eines `Stack`s für die Übergabe an den Client gesetzt. Anschließend werden die unterstützten Cipher Suiten und das Timeout der Session analog dem Client festgelegt. Es werden für diese Zwecke ebenfalls die optionalen Parameter für die Methoden in der Aufrufhierarchie hinzugefügt. Damit die Verwendung von ephimeralen elliptischen Kurven möglich ist, muss die `SSL_CTX_set_tmp_ecdh`-Funktion mit einer Kurve als Parameter aufgerufen werden. Im letzten Schritt wird das zu benutzende Serverzertifikat mit dem dazugehörigen Schlüssel festgelegt und eine Session-ID gesetzt. Diese ID bildet sich aus dem Namen der Serveranwendung. Mit diesem initialisierten Kontext kann ein `Ssl`-Objekt erstellt werden.

4.8.3 Verbindungsaufbau

Da sich in der Analyse zeigte, dass der Verbindungsaufbau selbst ordnungsgemäß funktioniert, wird darauf nicht weiter eingegangen. Stattdessen wird eine Netzwerkkommunikation in vier Stufen getestet. Zunächst wird getestet, ob generell eine Kommunikation mit den gegeben Zertifikaten stattfinden kann. Der zweite Schritt ist ein Test der Kommunikation zwischen dem OpenSSL.NET Server bzw. Client und dem Gegenpart des OpenSSL Kommandozeilentools. Danach wird die Kommunikation von OpenSSL.NET Client zum OpenSSL.NET Server durchgeführt. Im letzten Schritt wird OpenSSL.NET die Rolle eines Smart Meter Gateways spielen und ein Smart Meter baut die Verbindung auf.

Die zu verwendenden Zertifikate werden mittels OpenSSL erstellt. Es wird jeweils ein selbstsigniertes Zertifikat für Client und Server generiert. Die dazugehörigen Schlüssel werden mittels secp384r1-Kurve, die auch als die vom BSI vorgeschlagene NIST P-384 Kurve bekannt ist. Die Signatur wird mittels ECDSA mit SHA1 (ecdsa-with-SHA1) erstellt. Das Serverzertifikat wird als „server.pem“, der dazugehörige private Schlüssel als „server.key“, das Clientzertifikat als „client.pem“ und der dazugehörige private Schlüssel als „client.key“ gespeichert. Zuerst wird der Server über das Kommandozeilentool gestartet:

```
openssl.exe s_server -cert server.pem -key server.key -www -Verify 2
```

Des Argument „s_server“ gibt an, dass ein Server erstellt werden soll, mittels „cert“ und „key“ wird das Zertifikat mit dem privaten Schlüssel übergeben. „Verify“ gibt an, dass der Client sein Zertifikat zum Server zur Authentifizierung gesendet werden soll. Der Parameter „www“ sagt nur aus, dass ein Webserver erstellt wird, der mit dem HTTP-Protokoll arbeitet und Informationen zur Verbindung zurückgibt. Im Anschluss wird der Client ebenfalls über die Kommandozeile gestartet:

```
openssl.exe s_client -connect localhost:4433 -key client.key
            -cert client.pem
```

Das Argument „s_client“ gibt an, dass ein Client erstellt werden soll, der sich mittels „connect“ auf den angegeben Server verbindet. Der Server befindet sich im diesem Fall auf demselben Rechner, es wird der Standard-Port 4433 benutzt. Wie in der Abbildung 4.6 zu erkennen ist, wird erfolgreich eine Verbindung hergestellt. Die Zertifikate werden ausgetauscht und erfolgreich verifiziert. Es wurde eine TLS 1.2 Verbindung mit der „ECDHE-ECDSA-AES256-GCM-SHA384“ Cipher Suite hergestellt.

Abbildung 4.6 OpenSSL Server - OpenSSL Client

Im nächsten Schritt bleibt der Server bestehen. Es wird zusätzlich ein einfacher Client mit dem OpenSSL.NET-Wrapper geschrieben, der „GET / HTTP/1.0“ zum Server für die Abfrage der Verbindungsdaten sendet. Im Gegensatz zum Kommandozeilentool werden diesmal die unterstützten Cipher Suites eingeschränkt. Es werden ausschließlich die vier von BSI zugelassenen Suites (siehe Kapitel 3.1) verwendet. Die vom Server zurückgegeben Daten werden zur Anzeige gebracht. Der interessante Ausschnitt hierbei ist in Abbildung 4.7 zu sehen. Es ist erkennbar, dass tatsächlich nur diese vier Cipher Suites zugelassen sind und dass eben die vom BSI als „TLS_**ECDHE_ECDSA**_WITH_**AES_128**_CBC_**SHA256**“ bezeichnete Cipher Suite benutzt wird.

Abbildung 4.7 OpenSSL Server - OpenSSL.NET Client

Als nächstes wird mit OpenSSL.NET eine kleine Server Applikation entwickelt, die sich ebenfalls die vier Cipher Suites beschränkt. Kann eine Verbindung erfolgreich aufgebaut werden, wird über das HTTP-Protokoll eine HTML-Seite ausgegeben, die Informationen zur Verbindung preisgibt. Dazu gehören unter anderem das verwendete SSL/TLS-Protokoll und

die vom Server unterstützten Cipher-Suites. Es ist in Abbildung 4.8 erkennbar, dass das korrekte Protokoll TLS 1.2 ausgewählt wurde. Aufgrund der internen Verarbeitung gibt der Server falsche Angaben zum verwendeten Hash-Algorithmus aus. Wie jedoch in Abbildung 4.9 – ein Ausschnitt der Client-Informationen der gleichen Verbindung – zu sehen ist, wurde eine korrekte und unterstützte Cipher Suite ausgewählt und verwendet.

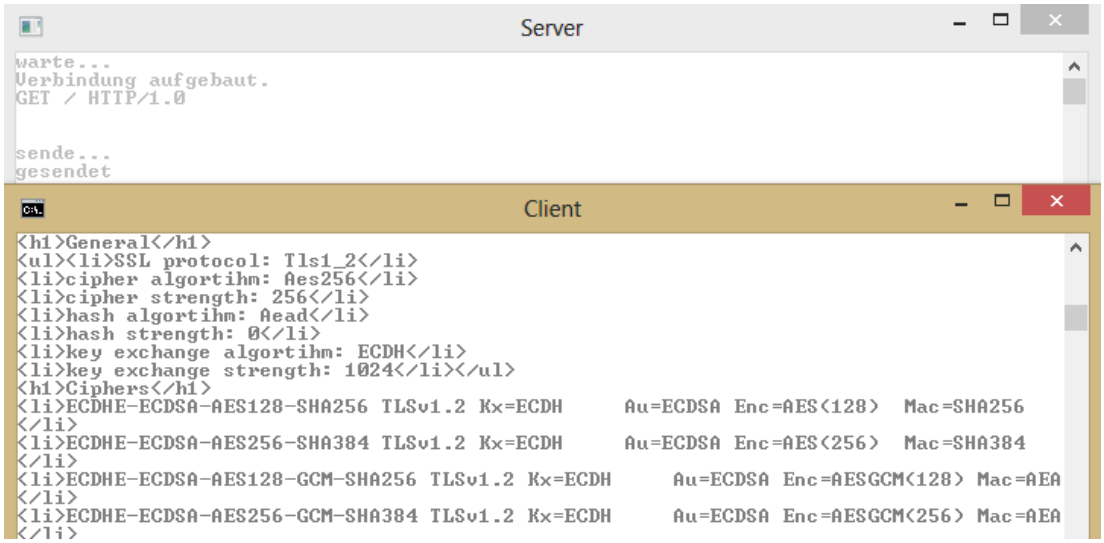

Abbildung 4.8 OpenSSL.NET Server - OpenSSL Client

Abbildung 4.9 OpenSSL.NET Server - OpenSSL Client (Cipher)

Ein vorletzter Test ist die Verbindung von OpenSSL.NET Client zum OpenSSL.NET Server. Beide werden in ihrer vorherigen Konfiguration verwendet. Der Verbindungsaufbau verläuft reibungsfrei und die Ausgabe der Session-Daten bestätigt erneut, dass das korrekte Protokoll mit der korrekten Session verwendet wird. In Abbildung 4.10 ist das Ergebnis dieser Verbindung sichtbar.

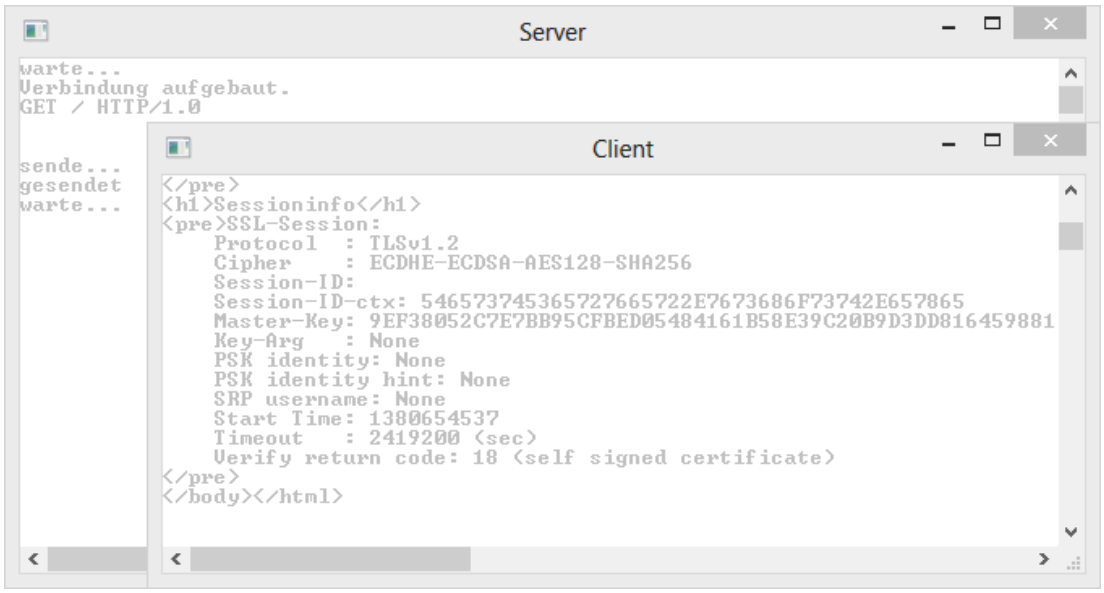

Abbildung 4.10 OpenSSL.NET Server - OpenSSL.NET Client

Da die vorherigen Tests erfolgreich waren, kann nun ein weiterer Test ausgeführt werden. Dieses Mal wird ein Szenario erstellt, wo ein OpenSSL.NET-Server die Rolle eines Smart Meter Gateways annimmt und auf eine Verbindung eines Smart Meters wartet. Die Clientsoftware des Smart Meters wurde unter Linux mit der OpenSSL-Bibliothek entwickelt. Während des Tests läuft diese in einer Virtuellen Maschine und hat daher die IP 192.168.190.159, welche über eine *Softwarebridge* mit dem Hochschulnetzwerk verbunden ist. Die Serversoftware läuft auf einem separaten Rechner und hat die IP 149.205.200.11. Beide Programme sind nach den Anforderungen des BSI für Smart Metering Systeme entwickelt worden. Zunächst kam es jedoch zu keinerlei Verbindung der beiden Rechner, was jedoch auf die Firewall-Einstellung zurückzuführen ist. Nach entsprechender Anpassung kann eine Verbindung zwischen beiden Systemen erfolgreich hergestellt werden. Der entstandene Netzwerkverkehr wurde mit der freien Wireshark-Software mitgeschnitten. Die Netzwerkpakete für einen erfolgreichen Handshake sind in der Abbildung 4.11 zu sehen. Es ist zu erkennen, dass der Client ausschließlich die vier vom BSI gestatteten Cipher Suites unterstützt werden. Zusätzlich ist noch die Cipher Suite „TLS_EMPTY_RENEGOTIATION_INFO_SCSV“ zu erkennen, hinter der jedoch keine Algorithmen stehen. Diese Suite wird dafür verwendet, um die Kompatibilität zwischen dem Client und einer schlechten Serverimplementierung, die die Verbindung abbrechen würde, wenn nach dem *Client Hello* weitere unbekannte Daten kommen würden, zu erhöhen. **/RFC5746/** Anschließend wurden die Zertifikate und Schlüssel ausgetauscht und mit dem Ändern der Cipher Suite wird der Handshake beendet. Es können anschließend die Daten entsprechend verschlüsselt übertragen werden.

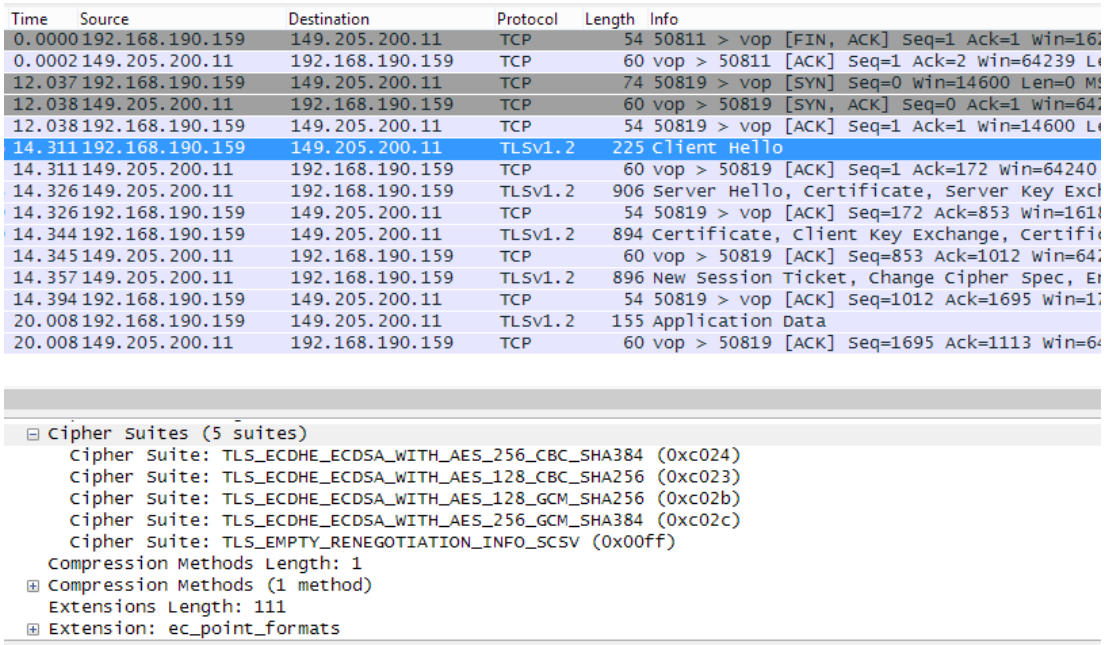

Abbildung 4.11 OpenSSL.NET Server - Linux Client (Wireshark-Mitschnitt)

5 Zusammenfassung

Um diese Arbeit zu einen Abschluss zu bringen, wird noch einmal auf die Tabelle 3-1 aus Kapitel 3.2 eingegangen. Diese wurde um eine weitere Spalte erweitert, um die Entwicklungsfortschritt von OpenSSL.NET in Tabelle 5-1 vergleichen zu können.

BSI-Anforderung	**OpenSSL**	**OpenSSL (erweitert)**	**OpenSSL.NET (vorher)**	**OpenSSL.NET (nachher)**
Brainpool P256r1	✗	✓	✗	✓
Brainpool P384r1	✗	✓	✗	✓
Brainpool P512r1	✗	✓	✗	✓
NIST P-256	✓	✓	✓	✓
NIST P-384	✓	✓	✓	✓
SHA-256	✓	✓	✓	✓
SHA-384	✓	✓	✓	✓
ECDSA	✓	✓	✓	✓
CMS	✓	✓	✗	✓
AES-GCM	✓	✓	✗	✓
AES-CBC-CMAC	✓	✓	✗	✓
AES Key Wrap	✓	✓	✗	✓
X.509 Zertifikat	✓	✓	✓	✓
TLS 1.2	✓	✓	✗	✓

Tabelle 5-1 Übersicht der Implementierung der BSI-Anforderungen in OpenSSL und OpenSSL.NET in der alten und neuen Version

Zusätzlich zu den bereits vorhandenen Funktionen ist OpenSSL.NET jetzt fähig, gesicherte Netzwerkkommunikation über TLS 1.2 zu führen. Dazu gehört auch die Möglichkeit unterstützte Cipher Suites nach Belieben einschränken zu können. Weiterhin ist es nun auch möglich, explizit zwischen den Protokollen SSL 2, SSL 3, TLS 1, TLS 1.1 und TLS 1.2 zu entscheiden und nur eines dieser zu unterstützen. Es ist gefordert die erzeugten Schlüssel für die Verschlüsselung der Daten verschlüsselt in einem CMS-Container sicher zu bewahren. Mit OpenSSL.NET ist es nun möglich, CMS-Container zu erstellen, diesen zu signieren, zu verschlüsseln, zu entschlüsseln und zu verifizieren. Die Schlüssel sollten außerdem mittels AES Key Wrap Verfahren verschlüsselt werden, was ebenfalls mit der modifizierten Version von OpenSSL.NET möglich ist. Auch im Zusammenhang mit CMS und der Verschlüsselung und Authentifizierung der Inhalte funktionieren auch die die AES-GCM bzw. AES-CBC mit dem CMAC-Verfahren. Eine Anforderung vom BSI an Smart Metering Systeme ist, dass die NIST-Kurven in den nächsten Jahren durch entsprechende Brainpoolkurven ersetzt werden sollen. Eine Besonderheit der angepassten OpenSSL.NET Version ist durch die Unterstützung von Brainpool vorrausschauend erfüllt. Die Brainpoolkurven können mit einer Einschränkung direkt aus OpenSSL.NET generiert und genutzt werden. In den erstellten Zertifikaten können keine Brainpool-Objekt-IDs der Kurven anstelle der Kurvenparameter eingebettet bzw. verwendet werden. Zusätzlich wird die Möglichkeit geboten, falls eine modifizierte Version von OpenSSL mit vorhandenen Brainpoolkurven vorliegt, diese für die Erstellung der Kurven zu nutzen. Auf diese Weise ist es auch möglich die Objekt-ID im Zertifikat zu benutzen, was die Dateigröße des Zertifikats wesentlich verkleinert. Zur Verschaulichung des Unterschieds wurden jeweils – abgesehen von der verwendeten Kurve für die Schlüssel – identische Zertifikate mit einer Brainpoolkurve im **Fehler! Verweisquelle konnte nicht gefunden erden.** und einer NIST P-384r1 Kurve im Anhang B erstellt.

Außer den hier in der Arbeit erwähnten Änderungen und Ergänzungen am OpenSSL.NET Projekt wurden weitere kleiner Anpassungen getätigt. So wurde beispielsweise vorher eine Funktion `SSL_CIPHER_name` benutzt, die so in der Form nicht existiert. Sie wurde durch die korrekte Funktion `SSL_CIPHER_get_name` ersetzt. Weiterhin wurden Quelldateien in der Hinsicht bearbeitet, dass einerseits ungenutzte und damit überflüssige `using`-Anweisungen entfernt wurden und andererseits fehlende `static`-Deklarierungen bei Klassen wie der zum Beispiel der `Random`-Klasse hinzugefügt. Das Kommandozeilentool wurde so modifiziert, dass in der Info-Ausgabe die in das Tool integrierten Befehle durch ein starkes Weiß hervorgehoben werden. Dies ist in der Abbildung 5.1 sichtbar. Das Kommandozeilen-Projekt wurde weitgehend nicht verändert, da für diese Funktionen des Tools das original OpenSSL Tool benutzt werden kann. Ziel von OpenSSL.NET ist es schließlich eine Schnittstelle für eigens geschriebene Programme zu liefern.

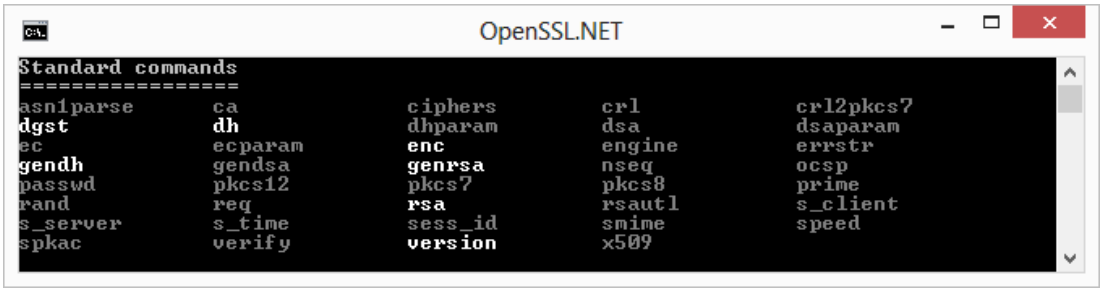

Abbildung 5.1 OpenSSL.NET Kommandozeilentool

In einer späteren Weiterentwicklung sollte die Überprüfung von Zertifikatsperrlisten mit integriert werden. Auch die Verwendung der `BIO`-Klasse kann erweitert werden, um den Komfort zu verbessern. So muss zurzeit der Dateimodus in Form einer Zeichenkette, wie zum Beispiel „w+“ für das Beschreiben und Lesen einer Datei, angegeben werden. Eine Möglichkeit ist es, für den Dateimodus, die im .NET Framework vorhandene `FileMode`-Enumeration zur Verwendung als Überladung anzubieten.

Literaturverzeichnis

/BSITR031091/ BUNDESAMT FÜR SICHERHEIT IN DER INFORMATIONSTECHNIK: *BSI TR-03109-1: Anforderungen an die Interoperabilität der Kommunikationseinheit eines intelligenten Messsystems.* , Version 1.0 Auflage (18. März 2013), BSI TR-03109-1.

/BSITR031093/ BUNDESAMT FÜR SICHERHEIT IN DER INFORMATIONSTECHNIK: *BSI TR-03109-3: Kryptographische Vorgaben für die Infrastruktur von intelligenten Messsystemen.* (18. März 2013), BSI TR-03109-3.

/BSITR031163/ BUNDESAMT FÜR SICHERHEIT IN DER INFORMATIONSTECHNIK: *BSI TR-03116-3: eCard-Projekte der Bundesregierung Teil 3.* (18. März 2013), BSI TR-03116-3.

/RFC5639/ M. LOCHTER; J. MERKLE: *RFC 5639: ECC Brainpool Standard Curves & Curve Generation.* (März 2010), ISSN: 2070-1721.

/RFC5652/ R. HOUSLEY: *Cryptographic Message Syntax (CMS).* (September 2009), RFC 5652.

/RFC5746/ E. RESCORLA; M. RAY; S. DISPENSA; N. OSKOV: *RFC5746: Transport Layer Security (TLS) Renegotiation Indication Extension.* (Februar 2010), ISSN 2070-1721.

/AGE13/ AG ENERGIEBILANZEN E.V.: Bruttostromerzeugung in Deutschland von 1990 bis 2012 nach Energieträgern (02.08.2013)
[http://www.ag-energiebilanzen.de/index.php?article_id=29&fileName=20130809_brd_stromerzeugung1990_2012.pdf] (Zugriff: 04.09.2013).

/BPDP05/ DR. MANFRED LOCHTER: ECC Brainpool Standard Curves and Curve Generation (19.10.2005)
[http://www.ecc-brainpool.org/download/Domain-parameters.pdf] (Zugriff: 10.07.2013).

/BPECC/ Brainpoolpatch für OpenSSL ec_curve.c
[https://github.com/openssl/openssl/blob/master/crypto/ec/ec_curve.c] (Zugriff: 09.07.2013).

/BPOBJ/ Brainpoolpatch für OpenSSL obj_mac.h
[https://github.com/openssl/openssl/blob/master/crypto/objects/obj_mac.h] (Zugriff: 09.07.2013).

/BSISMS13/ BSI: Smart Metering Systems
[https://www.bsi.bund.de/DE/Themen/SmartMeter/smartmeter_node.html] (Zugriff: 04.09.2013).

/BSITR13/ BSI TR-03109
[https://www.bsi.bund.de/DE/Themen/SmartMeter/TechnRichtlinie/TR_node.html] (Zugriff: 04.09.2013).

/ECCBP/ ECC Brainpool
[http://www.ecc-brainpool.org/] (Zugriff: 18.09.2013).

/MON13/ Mono-Project
[http://www.mono-project.com] (Zugriff: 27.08.2013).

/MSD13/ MSDN: Enum.HasFlag-Methode
[http://msdn.microsoft.com/de-de/library/vstudio/system.enum.hasflag.aspx] (Zugriff: 27.09.2013).

/MSDBOX/ MSDN: Boxing und Unboxing
[http://msdn.microsoft.com/de-de/library/yz2be5wk.aspx] (Zugriff: 27.09.2013).

/MSDNFLG/ MSDN: FlagsAttribute-Klasse
[http://msdn.microsoft.com/de-de/library/vstudio/system.flagsattribute.aspx] (Zugriff: 14.08.2013).

/MSDNNET45/ MSDN: What's New in the .NET Framework 4.5
[http://msdn.microsoft.com/en-us/library/ms171868.aspx] (Zugriff: 31.08.2013).

/MSDNUFP/ MSDN: UnmanagedFunctionPointerAttribute-Klasse
[http://msdn.microsoft.com/de-de/library/vstudio/system.runtime.interopservices.unmanagedfunctionpointerattribute%28v=vs.100%29.aspx] (Zugriff: 15.09.2013).

/MSDNVS11/ MSDN: Visual Studio 2012 - Kompatibilität
[http://msdn.microsoft.com/de-de/library/vstudio/hh266747.aspx] (Zugriff: 10.09.2013).

/MSDPRPR/ MSDN: C#-Präprozessordirektiven
[http://msdn.microsoft.com/de-de/library/ed8yd1ha.aspx] (Zugriff: 16.09.2013).

/MSSLN2/ Microsoft Support Lifecycle
[http://support.microsoft.com/lifecycle/default.aspx?LN=de&p1=8291] (Zugriff: 30.08.2013).

/ONETCC/ OpenSSL.NET - Code commits
[http://sourceforge.net/p/openssl-net/code/commit_browser] (Zugriff: 28.08.2013).

/ONETL/ OpenSSL.NET: License
[http://openssl-net.sourceforge.net/LICENSE] (Zugriff: 28.08.2013).

/OSSLA/ OpenSSL: About
[http://openssl.net/about/] (Zugriff: 27.08.2013).

/OSSLCL/ OpenSSL: ChangeLog
[http://www.openssl.org/news/changelog.html] (Zugriff: 12.09.2013).

/OSSLCMS1/ OpenSSL: CMS_sign(3)
[http://openssl.net/docs/crypto/CMS_sign.html] (Zugriff: 26.09.2013).

/OSSLCMS2/ OpenSSL: CMS_verify(3)
[http://openssl.net/docs/crypto/CMS_verify.html] (Zugriff: 26.09.2013).

/OSSLCMS3/ OpenSSL: CMS_encrypt(3)
[http://openssl.net/docs/crypto/CMS_encrypt.html] (Zugriff: 26.09.2013).

/OSSLCMS4/ OpenSSL: CMS_decrypt(3)
[http://openssl.net/docs/crypto/CMS_decrypt.html] (Zugriff: 26.09.2013).

/OSSLCMSCLI/ OpenSSL: cms(1)
[https://www.openssl.org/docs/apps/cms.html] (Zugriff: 26.09.2013).

/OSSLCTX/ OpenSSL: SSL_CTX_new(3)
[http://www.openssl.org/docs/ssl/SSL_CTX_new.html] (Zugriff: 24.09.2013).

/OSSLEVPDI/ OpenSSL: EVP_DigestInit(3)
[http://www.openssl.org/docs/crypto/EVP_DigestInit.html] (Zugriff: 11.09.2013).

/OSSLL/ OpenSSL: License
[http://www.openssl.org/source/license.html] (Zugriff: 27.08.2013).

/OSSLS/ OpenSSL: Source
[http://openssl.net/source/] (Zugriff: 27.08.2013).

/SLP13/ Shining Light Productions - Win32 OpenSSL
[http://slproweb.com/products/Win32OpenSSL.html] (Zugriff: 27.08.2013).

Anhang A

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 3 (0x3)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN=TEST root, C=DE, L=Merseburg, ST=Sachsen Anhalt, O=HS
            Merseburg
        Validity
            Not Before: Sep  2 16:27:23 2013 GMT
            Not After : Sep  2 16:27:23 2014 GMT
        Subject: CN=localhost, C=DE, L=Merseburg, ST=Sachsen Anhalt, O=HS
            Merseburg
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:06:0d:d9:ad:7b:48:51:70:c2:62:e8:a0:f3:24:
                    d0:bd:8a:0d:e9:e0:38:e5:cd:a0:83:36:1c:da:fc:
                    26:08:39:4a:ee:5c:af:53:4b:1e:80:67:31:79:a9:
                    65:5d:61:5e:0b:be:3d:66:7c:b1:40:d2:06:8d:f8:
                    1b:84:89:da:e7
                Field Type: prime-field
                Prime:
                    00:a9:fb:57:db:a1:ee:a9:bc:3e:66:0a:90:9d:83:
                    8d:72:6e:3b:f6:23:d5:26:20:28:20:13:48:1d:1f:
                    6e:53:77
                A:
                    7d:5a:09:75:fc:2c:30:57:ee:f6:75:30:41:7a:ff:
                    e7:fb:80:55:c1:26:dc:5c:6c:e9:4a:4b:44:f3:30:
                    b5:d9
                B:
                    26:dc:5c:6c:e9:4a:4b:44:f3:30:b5:d9:bb:d7:7c:
                    bf:95:84:16:29:5c:f7:e1:ce:6b:cc:dc:18:ff:8c:
                    07:b6
                Generator (uncompressed):
                    04:8b:d2:ae:b9:cb:7e:57:cb:2c:4b:48:2f:fc:81:
                    b7:af:b9:de:27:e1:e3:bd:23:c2:3a:44:53:bd:9a:
                    ce:32:62:54:7e:f8:35:c3:da:c4:fd:97:f8:46:1a:
                    14:61:1d:c9:c2:77:45:13:2d:ed:8e:54:5c:1d:54:
                    c7:2f:04:69:97
                Order:
                    00:a9:fb:57:db:a1:ee:a9:bc:3e:66:0a:90:9d:83:
                    8d:71:8c:39:7a:a3:b5:61:a6:f7:90:1e:0e:82:97:
                    48:56:a7
                Cofactor:
                    00:8b:d2:ae:b9:cb:7e:57:cb:2c:4b:48:2f:fc:81:
                    b7:af:b9:de:27:e1:e3:bd:23:c2:3a:44:53:bd:9a:
                    ce:32:62
        X509v3 extensions:
            X509v3 Key Usage:
                Digital Signature, Data Encipherment, Key Agreement
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
    Signature Algorithm: ecdsa-with-SHA256
         30:44:02:20:4f:6e:c1:be:51:8a:05:da:d7:35:99:0f:4f:22:
         27:c3:1d:b4:c2:1b:50:c6:c5:83:97:63:45:5e:99:66:76:34:
         02:20:4b:72:32:7f:73:11:93:3f:ae:7f:98:20:4c:ce:eb:ed:
         23:02:ba:62:e8:01:f9:81:fd:de:c9:07:c4:38:e9:ba
```

Anhang B

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 3 (0x3)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN=TEST root, C=DE, L=Merseburg, ST=Sachsen Anhalt, O=HS
            Merseburg
        Validity
            Not Before: Sep  4 12:54:23 2013 GMT
            Not After : Sep  4 12:54:23 2014 GMT
        Subject: CN=localhost, C=DE, L=Merseburg, ST=Sachsen Anhalt, O=HS
            Merseburg
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (384 bit)
                pub:
                    04:92:17:68:d0:c5:89:91:6c:2c:2f:02:3e:c3:14:
                    65:65:91:ea:e0:1e:66:8c:c5:43:cd:76:17:8a:8a:
                    a3:f7:a1:22:55:07:b4:8f:ff:15:d9:2d:1b:6e:2c:
                    92:f4:f9:1e:f1:54:47:25:3b:78:0e:41:7d:53:8d:
                    91:57:b4:92:17:43:53:25:ce:de:71:50:ee:6f:f6:
                    c9:6b:f9:f3:64:66:de:a3:fa:a2:57:b2:22:b5:c9:
                    51:4b:02:24:ae:14:66
                ASN1 OID: secp384r1
        X509v3 extensions:
            X509v3 Key Usage:
                Digital Signature, Data Encipherment, Key Agreement
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
    Signature Algorithm: ecdsa-with-SHA256
         30:66:02:31:00:ac:96:60:b9:89:1b:51:15:21:32:e2:d7:d6:
         91:9a:4d:99:0c:7d:46:93:a5:44:fc:c0:8a:14:b6:c7:8e:08:
         a9:ca:db:b9:50:d8:86:69:6f:37:ca:43:fa:af:a9:15:02:02:
         31:00:a9:38:26:89:50:de:f2:47:10:51:a8:f2:62:9f:44:01:
         6e:54:55:9e:5f:0c:a4:56:8e:52:19:ec:49:b5:1e:cb:cc:1a:
         c4:e4:af:93:65:26:26:f9:d5:75:e3:71:1e:37
```

Eidesstattliche Erklärung

Ich versichere eidesstattlich durch eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe und die den benutzten Quellen wörtliche oder inhaltlich entnommene Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____

Erik Groß