Universität Hamburg

Fachbereich Informatik

Entwurf vom
16. Oktober 2015

Bachelorarbeit

Funktionsweise, Angriffe und Abwehrmechanismen von SSL/TLS

vorgelegt von

Tom Petersen

geb. am 13. Dezember 1990 in Hannover

Matrikelnummer 6359640

Studiengang Informatik

eingereicht am 16. Oktober 2015

Betreuer: Dipl.-Inf. Ephraim Zimmer

Erstgutachter: Prof. Dr.-Ing. Hannes Federrath

Zweitgutachter: Dr. Dominik Herrmann

Aufgabenstellung

Die Protokollfamilie SSL/TLS umfasst Techniken zum Schutz von Kommunikationsdaten in IP-basierten Netzen. Ihre weite Verbreitung und Wichtigkeit für die IT-Sicherheit ist historisch gewachsen, und ihr Einsatz erstreckt sich über mittlerweile weit mehr Protokolle der Anwendungsschicht, als nur das ursprünglich anvisierte HTTP. Diese weite Verbreitung hat zwei wesentliche Konsequenzen. Zum einen wurden sowohl die Spezifikation der Protokollfamilie als auch praktische Implementierungen von SSL/TLS Gegenstand zahlreicher Angriffe. Zum zweiten sind ein grundlegendes Verständnis der Funktionsweise von SSL/TLS und der erwähnten Angriffe obligatorisch bei der Entwicklung und Implementierung von verteilter Software, Internetdiensten und Protokollimplementierungen auf der Anwendungsschicht, die mittels SSL/TLS abgesichert werden sollen.

In dieser Bachelorarbeit soll unter Einbeziehung aktueller Entwicklungen und Forschungsergebnisse die Funktionsweise von SSL/TLS, bedeutende Angriffe auf diese Protokollfamilie sowie daraus erarbeitete Anpassungen der Protokollspezifikation und Abwehrmechanismen erläutert und speziell für den Einsatz in der Hochschullehre aufbereitet werden. Darüber hinaus soll ein modular aufgebautes Tool zur Veranschaulichung der SSL/TLS-Funktionsweise sowie deren Angriffe und Abwehrmechanismen entwickelt und prototypisch umgesetzt werden. Der Fokus des Tools liegt in der Demonstration von SSL/TLS und dessen Schwächen mit beliebiger Verständnisvertiefung, sollte allerdings auch um weitere IT-Sicherheitsprotokolle erweiterbar sein.

Zusammenfassung

Zu schreiben...

Abstract schreiben

Inhaltsverzeichnis

1 Einleitung

TLS ist eines der bedeutendsten Sicherheitsprotokolle, das heute verwendet wird. Viele andere Protokolle setzen darauf auf, um ihre Kommunikation abzusichern. Aufgrund seiner Bedeutung wurde es im Laufe seiner Entwicklung oft untersucht und angegriffen. Dabei sind viele einfache und elegante Angriffe gefunden worden, die zeigen, wie wirksam die kleinsten Schwächen in Protokollen ausgenutzt werden können. Ebenso bieten Änderungen in der TLS-Spezifikation auch Beispiele für erfolgreiche Gegenmaßnahmen, die diese Angriffe verhindern.

In dieser Arbeit soll das TLS-Protokoll auf seine Eignung für den Einsatz in der Hochschullehre überprüft und dafür aufbereitet werden. Neben einer ausführlichen Betrachtung der Funktionsweise von TLS und existierenden Angriffen gegen die Protokollfamilie werden Empfehlungen für die Nutzung in der Lehre herausgearbeitet. Außerdem wird eine interaktive Anwendung entwickelt, die die Protokollabläufen während einer TLS-Verbindung simuliert und damit ein vertieftes Verständnis von TLS unterstützt.

In Kapitel 2 werden in TLS zur Anwendung kommende kryptographische Verfahren eingeführt und die in dieser Arbeit verwendete Notation erläutert.
Der erste Teil der Arbeit behandelt die Grundlagen von TLS und entdeckter Angriffe. Das Kapitel 3 beschäftigt sich ausführlich mit der Funktionsweise von TLS anhand der aktuellen Protokollversion TLS 1.2. In Kapitel 4 wird auf bisher entdeckte Angriffe gegen SSL und TLS und getroffene Abwehrmechanismen eingegangen.
Im zweiten Teil der Arbeit geht es um den Einsatz von TLS in der Hochschullehre. Dazu werden in Kapitel 5 Schwerpunkte für die Lehre herausgearbeitet, die am Beispiel von TLS erläutert werden können. Außerdem werden didaktische Grundlagen für die im Rahmen dieser Bachelorarbeit entwickelte Protokollsimulation gelegt. In Kapitel 6 wird dann der Aufbau der Anwendung und der Erweiterung um TLS beschrieben.

Ein Großteil dieser Arbeit stützt sich direkt auf die TLS 1.2-Spezifikation in [DR08]. Für die Grundlagen zu verwendeten kryptographischen Verfahren wurden [Sch06] und [FSK10] zu Hilfe genommen.

2 Einführung

In diesem Kapitel soll ein kurzer Überblick über die im Laufe der Arbeit relevanten kryptographischen Verfahren gegeben werden, sowie die verwendete Notation vorgestellt werden.

2.1 Kryptographische Verfahren

2.1.1 Symmetrische Kryptographie

Symmetrische Kryptographie beschreibt Verfahren, bei denen bei der Ver- und Entschlüsselung der gleiche (oder ein aus dem anderen Schlüssel leicht berechenbarer) Schlüssel verwendet wird. Vor einer Kommunikation müssen beide Kommunikationspartner im Besitz dieses Schlüssels sein, ihn also über einen sicheren Kanal ausgetauscht haben.
Es gilt:

$$E_K(M) = C$$
$$D_K(C) = M$$

Hierbei steht M für die Nachricht, K für den Schlüssel, der verwendet wird, C für den Chiffretext und E bzw. D für die Ver- bzw. Entschlüsselung. Der Vorgang der symmetrischen Ver- und Entschlüsselung wird in Abbildung 2.1 dargestellt.

Symmetrischen Chiffren lassen sich in Strom- und Blockchiffren unterteilen.

Stromchiffren

Stromchiffren sind symmetrische Verschlüsselungsalgorithmen, die Klartexte bitweise zu Chiffretexten konvertieren. Die einfachste Möglichkeit ist die XOR-Verknüpfung der Klartextbits mit einem schlüsselabhängig generierten Bitstrom (Schlüsselstrom). Durch erneute XOR-Verknüpfung mit dem selben Schlüsselstrom auf der Empfängerseite lässt sich der Klartext zurückerhalten. Ein Beispiel für eine heute verwendete Stromchiffre ist RC4 [Sch06].

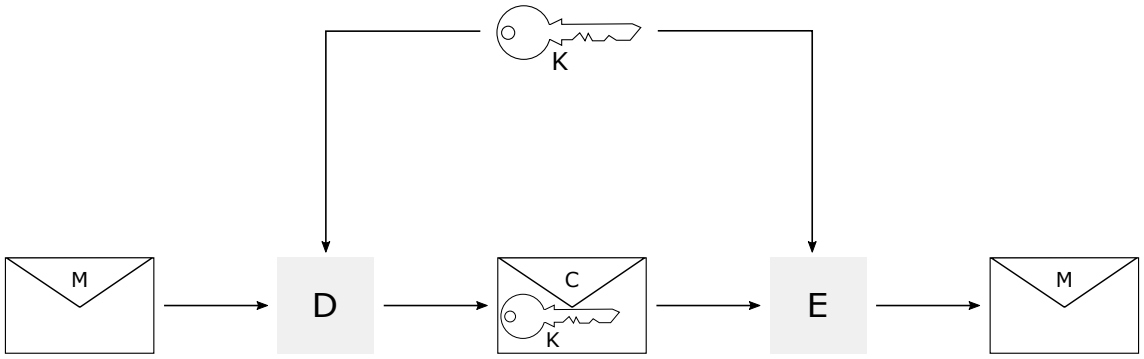

Abbildung 2.1: Symmetrische Kryptographie

Blockchiffren

Bei Blockchiffren handelt es sich um symmetrische Verschlüsselungsalgorithmen, die Nachrichten in Blöcken fester Größe verschlüsseln. Beispiele für heute verwendete Blockchiffren sind AES oder Twofish.

Da eine Blockchiffre immer nur einen Block verschlüsseln kann, muss festgelegt werden, wie mit mehreren Blöcken verfahren werden soll. Die Beschreibung eines solchen Verfahrens wird Betriebsmodus genannt.

Der einfachste Modus ist der ECB-Modus (Electronic Codebook). In diesem Modus wird jeder Block einzeln und unabhängig von anderen Blöcken verschlüsselt. Dieser Modus ist als unsicher zu betrachten, da die Verschlüsselung gleicher Klartextblöcke mit gleichem Schlüssel immer zu dem gleichen Chiffretextblock führt und ein Angreifer außerdem nach Belieben einzelne Blöcke entfernen, hinzufügen oder austauschen kann, ohne dass dies zwingend bemerkt wird.

Ein Beispiel für einen sichereren Modus ist CBC (Cipher Block Chaining). Hierbei erfolgt eine XOR-Verknüpfung des zuletzt erhaltenen Chiffretextblocks mit dem nächsten Klartextblock vor seiner Verschlüsselung. Zusätzlich wird für den ersten Klartextblock ein zusätzlicher, zufällig gewählter Block, der sogenannte Initialisierungsvektor (IV), benötigt [Sch06].

Einen Sonderfall stellen die AEAD-Chiffren (Authenticated Encryption with Associated Data) dar. Hierbei handelt es sich um Betriebsmodi, die ohne zusätzlichen Message Authentication Code (siehe Abschnitt 2.1.5) Authentizität und Integrität bereitstellen. Beispiele für solche Modi sind CCM (Counter with CBC-MAC) oder GCM (Galois/Counter Mode).

2.1.2 Asymmetrische Kryptographie

Asymmetrische Kryptographie (oftmals auch Public-Key-Kryptographie) beschreibt Verfahren, bei denen bei der Ver- und Entschlüsselung verschiedene Schlüssel verwendet werden. Diese lassen sich nicht aus dem jeweils anderen Schlüssel berechnen. Daher kann ein Empfänger seinen öffentlichen Schlüssel bekanntgeben. Nachrichten, die mit diesem Schlüssel verschlüsselt werden, kann ein Angreifer dennoch nicht lesen, da nur der Empfänger im Besitz seines geheimen Schlüssels ist. Es gilt:

$$E_{K_{\text{public}}}(M) = C$$
$$D_{K_{\text{private}}}(C) = M$$

Hierbei steht K_{public} für den öffentlichen und K_{private} für den geheimen Schlüssel des Empfängers. Der Vorgang der asymmetrischen Ver- und Entschlüsselung ist in Abbildung 2.2 abgebildet.

Weiterhin können asymmetrische Verfahren auch zur Signierung von Nachrichten genutzt werden, indem der Verfasser die Nachricht (bzw. aus Effizienzgründen in der Praxis einen Hashwert der Nachricht) mit seinem geheimen Schlüssel verschlüsselt und an die Nachricht anhängt. Empfänger können nun durch Entschlüsselung mit dem öffentlichen Schlüssel und Vergleich mit der empfangenen Nachricht (bzw. ihrem Hashwert) die Signatur überprüfen.

Durch asymmetrische Kryptographie lässt sich das bei symmetrischen Algorithmen bestehende Problem des Schlüsselaustauschs durch Veröffentlichung des öffentlichen Schlüssels leicht lösen. Das Problem, das hierbei entsteht, ist jedoch die Identität des Besitzers eines öffentlichen Schlüssels sicherzustellen.

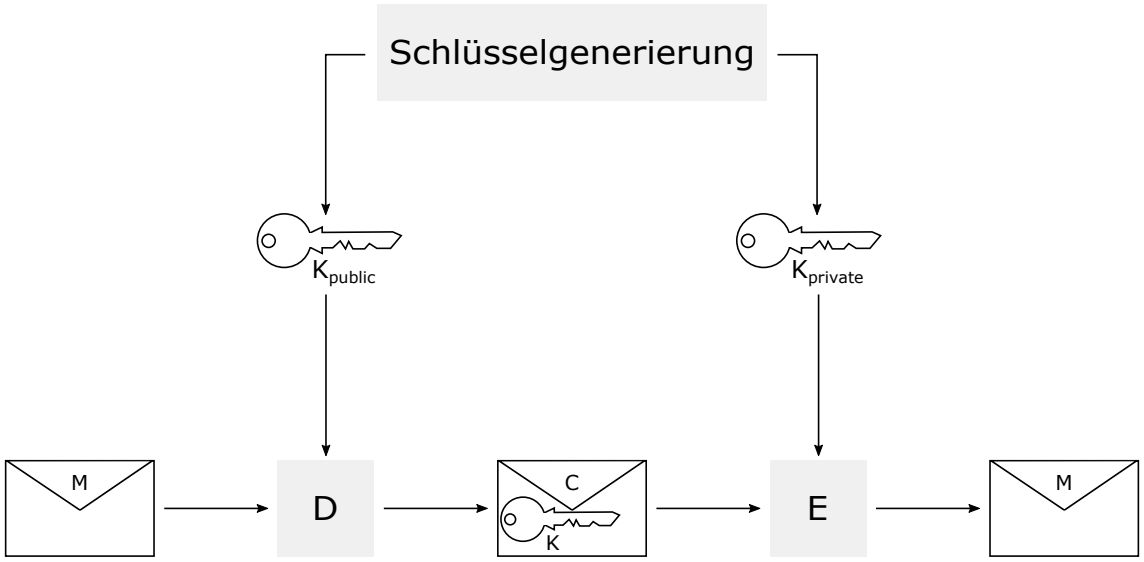

Abbildung 2.2: Asymmetrische Kryptographie

Ein Beispiel für einen asymmetrischen Algorithmus ist das RSA-Verfahren [Sch06].

2.1.3 Diffie-Hellmann-Verfahren

Das Diffie-Hellmann-Verfahren (DH-Verfahren) ist ein ebenfalls auf geheimen und öffentlichen Schlüssel basierendes Verfahren, das jedoch nicht der Ver- bzw. Entschlüsselung, sondern lediglich der Schlüsselvereinbarung dient. Es ermöglicht zwei Kommunikationspartnern einen Schlüssel zu erzeugen, ohne dass dieser direkt gesendet werden muss.

Die Partner A und B einigen sich auf eine Gruppe primer Ordnung p und eine Zahl g, die die Gruppe erzeugt. Zusätzlich wählt jeder Partner eine große, zufällige Zahl X als privaten Schlüssel. Dann berechnet jeder Partner seinen öffentlichen Schlüssel Y folgendermaßen:

$$\begin{aligned} Y_A &= g^{X_A} \mod n \text{ bzw.} \\ Y_B &= g^{X_B} \mod n \end{aligned}$$

Diese Werte werden nun an den anderen Partner gesendet und der gemeinsame Schlüssel K kann berechnet werden:

$$\begin{aligned} K = Y_B^{X_A} \mod n = (g^{X_B})^{X_A} \mod n = g^{X_A X_B} \mod n \text{ bzw.} \\ K = Y_A^{X_B} \mod n = (g^{X_A})^{X_B} \mod n = g^{X_A X_B} \mod n \end{aligned}$$

Ein Angreifer, der lediglich im Besitz von Y_A und Y_B ist kann den Schlüssel nicht berechnen [Sch06].

2.1.4 Hashfunktion

Eine Hashfunktion ist eine Funktion, die eine Eingabe variabler Länge auf einen String fester Länge abbildet.

In der Kryptographie werden insbesondere Einweg-Hashfunktionen eingesetzt. Bei dieser Art von Hashfunktionen ist es leicht, aus einer Eingabe den Hashwert zu berechnen, jedoch sehr schwer, zu einem gegebenen Hashwert eine Eingabe zu finden, die auf diesen Wert abgebildet wird [Sch06].
Beispiele für heute verwendete Hashfunktionen sind MD5 und SHA256.

2.1.5 Message Authentication Code

Ein Message Authentication Code (MAC) ist ein Verfahren, das der Authentizität und dem Schutz der Integrität einer Nachricht dient. Dazu wird vom Sender aus einem geheimen Schlüssel K und der Nachricht M eine Art Prüfsumme generiert und zusammen mit der Nachricht versendet. Der Empfänger kann den MAC überprüfen, wenn er im Besitz des gleichen geheimen Schlüssels ist, und so sicherstellen, dass die Nachricht nicht verändert wurde. Ein Beispiel für einen solchen MAC ist der auch in TLS verwendete HMAC [Sch06, FSK10].

2.2 Verwendete Notationen

In den folgenden Kapiteln werden die unten stehenden Notationen verwendet. Aus anderen Veröffentlichungen entnommene Passagen wurden teilweise geringfügig angepasst, um diesen Notationen zu folgen.

$|B|$ steht für die Länge einer Zeichenkette B

$A \oplus B$ entspricht der bitweisen XOR-Verknüpfung zweier Zeichenketten A und B

$A + B$ steht für die Konkatenation zweier Zeichenketten A und B

$0x\ldots$ entspricht einer Zahl im hexadezimalen System

3 Funktionsweise von TLS

SSL (Secure Socket Layer) bzw. TLS (Transport Layer Security) ist ein zustandsbehaftetes Protokoll, das auf dem TCP-Protokoll[1] der Transportschicht des ISO/OSI-Schichtenmodells aufbaut.

Hauptaufgaben von TLS sind Authentifikation der Kommunikationspartner, Verschlüsselung der Kommunikation sowie die Sicherstellung der Integrität der übertragenen Nachrichten [Mey14]. Dazu läuft die Kommunikation über TLS in zwei Phasen ab: Zu Beginn wird eine sichere Verbindung durch Festlegung der verwendeten kryptographischen Verfahren und des Schlüsselmaterials hergestellt. Danach können Daten transparent für Anwendungen und auf TLS aufbauende Protokolle über diese Verbindung gesendet werden.
Einige Beispiele für solche Protokolle und Anwendungen der Anwendungsschicht, die TLS nutzen, sind:

HTTPS für die Datenübertragung, zumeist für die Auslieferung von Webseiten genutzt.

FTPS für die Dateiübertragung.

SMTP für das Senden und Weiterleiten von E-Mails (als SMTPS oder per STARTTLS[2]).

IMAP für den Zugriff auf E-Mails auf Mailservern (als IMAPS oder per STARTTLS[2]).

POP3 für den Abruf von E-Mails von Mailservern (als POP3S oder per STARTTLS[2]).

OpenVPN, eine verbreitete VPN-Software.

SSL wurde von der Firma Netscape entwickelt und zuerst in ihrem Browser, dem Netscape Navigator, verwendet. Nach mehreren neuen Protokollversionen und nachdem es starke Verbreitung gefunden hatte, wurde es durch die IETF als TLS 1.0 standardisiert (TLS 1.0 entspricht SSL 3.1). Aktuell ist die TLS-Version 1.2 und an Version 1.3 wird gearbeitet[3].
Inzwischen ist TLS das „gegenwärtig meistverwendete Verschlüsselungsprotokoll im Internet“ [Sch09]. Gründe hierfür sind dem Autor zufolge insbesondere die leichte Integrierbarkeit in bestehende Strukturen, die „[im Gegensatz zu IPSec] deutlich schnörkelloser[e] und einfacher[e]“ [Sch09] Protokollspezifikation und auch die marktreife Verfügbarkeit in den frühen 90er Jahren.

In den folgenden Abschnitten wird die Funktionsweise von TLS anhand der aktuellen Version 1.2 im Detail betrachtet. Dazu wird zunächst ein Überblick über die Teilprotokolle gegeben und diese anschließend genauer beleuchtet. Abschließend werden kurz Unterschiede zu anderen Protokollversionen betrachtet und auf bestehende Implementierungen der Spezifikation eingegangen.

1. Es existiert auch noch DTLS (Datagram Transport Layer Security), ein zu TLS ähnliches Protokoll, das auf UDP aufsetzt. Dieses Protokoll wird im Rahmen dieser Arbeit jedoch nicht weiter behandelt.
2. SMTPS/IMAPS/POP3S beginnen die TLS-Verbindung bereits direkt nach dem Verbindungsaufbau und laufen über einen anderen Serverport, um dieses Verhalten zu erzwingen. STARTTLS ist ein Kommando, das nach Verbindungsaufbau gesendet werden kann, um eine TLS-Verbindung zu initiieren.
3. Im weiteren Verlauf dieser Arbeit wird der Einfachheit halber lediglich von TLS gesprochen. Bei etwaigen Unterschieden zwischen den Protokollversionen wird explizit auf diese eingegangen.

3.1 Teilprotokolle

Die Informationen in diesem und den folgenden Abschnitt stammen überwiegend aus der TLS 1.2-Spezifikation [DR08]. Für einen ersten Überblick wurde [Eck13] genutzt.

TLS besteht aus zwei Schichten. Eine Übersicht über diese Schichten bietet Abbildung 3.1.
In der oberen Schicht sind vier Teilprotokolle spezifiziert: das Handshake-Protokoll, das zum Aushandeln von kryptographischen Algorithmen und zur Vereinbarung von Schlüsselmaterial dient, das ChangeCipherSpec-Protokoll, das den Beginn der Nutzung dieser Algorithmen regelt, das Alert-Protokoll, das das Versenden von Fehlerinformationen in der Verbindung übernimmt und das ApplicationData-Protokoll, das für den Austausch von Anwendungsdaten genutzt wird. Auf diese Protokolle soll später eingegangen werden.
In der unteren Schicht befindet sich das Record-Protokoll, das die Daten von den Teilprotokollen der oberen Schicht entgegennimmt, verarbeitet und dann an tiefere Netzwerkschichten weitergibt.

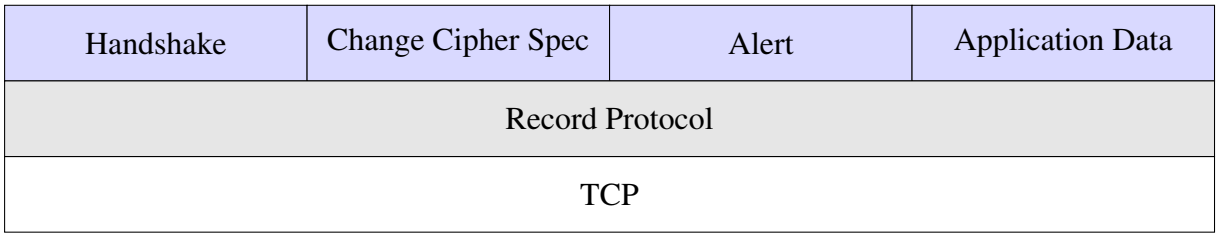

Abbildung 3.1: Überblick über die TLS-Protokollhierarchie

3.2 Record-Protokoll

Die zu sendenden Protokolldaten werden von dem Record-Protokoll in maximal 2^{14} Byte große Pakete fragmentiert und optional komprimiert. Danach wird je nach während des Handshakes verhandelten kryptographischen Funktionen (vgl. Abschnitt 3.4) die Integrität der Daten durch Berechnen und Anhängen eines MACs gesichert und die Nachricht danach zusammen mit dem MAC verschlüsselt (MAC-then-Encrypt, vgl. hierzu auch Abschnitt 5.1.5). Auf diese Schritte wird im Folgenden genauer eingegangen.

Daten, die von einer höheren Schicht entgegengenommen werden, werden zu Beginn in ein `TLSPlaintext`-Objekt verpackt.

```
struct {
   ContentType type;
   ProtocolVersion version;
   uint16 length;
   opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

Der `Contenttype` steht für den Protokolltyp der Nachricht: ChangeCipherSpec-Protokoll (20), Alert-Protokoll (21), Handshake-Protokoll (22) oder ApplicationData-Protokoll (23). Die `ProtocolVersion` besteht aus zwei Bytes für die über- und untergeordnete Protokollnummer (z.B. (3,3) für TLS 1.2). Im `fragment` werden die zu übertragenden Daten gespeichert.

Danach werden die Daten optional durch den während des Handshakes vereinbarten Kompressionsalgorithmus komprimiert und in ein `TLSCompressed`-Objekt überführt.

```
struct {
    ContentType type;       /* same as TLSPlaintext.type */
    ProtocolVersion version;/* same as TLSPlaintext.version */
    uint16 length;
    opaque fragment[TLSCompressed.length];
} TLSCompressed;
```

Anschließend wird die Integrität der Daten bei Nutzung einer Cipher-Suite mit Strom- oder Blockverschlüsselungsalgorithmus (vgl. Abschnitt 3.9) mit einem MAC geschützt, der folgendermaßen berechnet wird:

```
MAC(MAC_write_key, seq_num +
                   TLSCompressed.type +
                   TLSCompressed.version +
                   TLSCompressed.length +
                   TLSCompressed.fragment);
```

Die jeweils für jeden Kommunikationspartner unabhängig fortlaufende Sequenznummer `seq_num` dient hierbei zur Verhinderung von Replay-Angriffen, also der erneuten Sendung von mitgelesenen Paketen durch einen Angreifer. Auf die Berechnung von `MAC_write_key` wird in Abschnitt 3.3 eingegangen.
Bei den in TLS verwendeten Cipher-Suites wird das HMAC-Verfahren zur Berechnung des MACs genutzt. Details zu diesem Verfahren sind in [KBC97] zu finden. Die hierbei verwendete Hashfunktion wird in der Cipher-Suite angegeben.

Danach wird das `TLSCompressed`-Objeckt in ein `TLSCiphertext`-Objekt überführt und dann versendet.

```
struct {
    ContentType type;       /* same as TLSCompressed.type */
    ProtocolVersion version;/* same as TLSCompressed.version */
    uint16 length;
    select (SecurityParameters.cipher_type) {
        case stream: GenericStreamCipher;
        case block:  GenericBlockCipher;
        case aead:   GenericAEADCipher;
    } fragment;
} TLSCiphertext;
```

Abhängig von dem verwendeten Verschlüsselungsverfahren sehen diese Nachrichten unterschiedlich aus. Abbildung 3.2 bietet einen Überblick über diese verschiedenen Nachrichten. Dabei sind die fett umrahmten Felder diejenigen, die mit dem jeweiligen Algorithmus verschlüsselt werden.

Bei Stromchiffren wird der MAC zusammen mit den Daten (`TLSCompressed.fragment`) verschlüsselt und übertragen.

Bei Blockchiffren werden die Daten zusammen mit dem MAC zuerst mit Padding versehen, um ein Vielfaches der Blocklänge als Nachrichtenlänge zu erhalten, wobei jedes Padding-Byte die Paddinglänge als Wert enthält und die Länge zusätzlich als Byte an die Nachricht angehängt wird. Folglich gilt: $|\text{daten}| + |\text{mac}| + |\text{padding}| + 1 = k * \text{blocklänge}, k \in \mathbb{N}$.
Für Blockchiffren im CBC-Modus wird ein IV benötigt (siehe Abschnitt 2.1.1). Dieser wird

für jede Nachricht zufällig generiert. Dann werden Daten, MAC, Padding und Paddinglänge zusammen verschlüsselt und mit dem IV versendet.

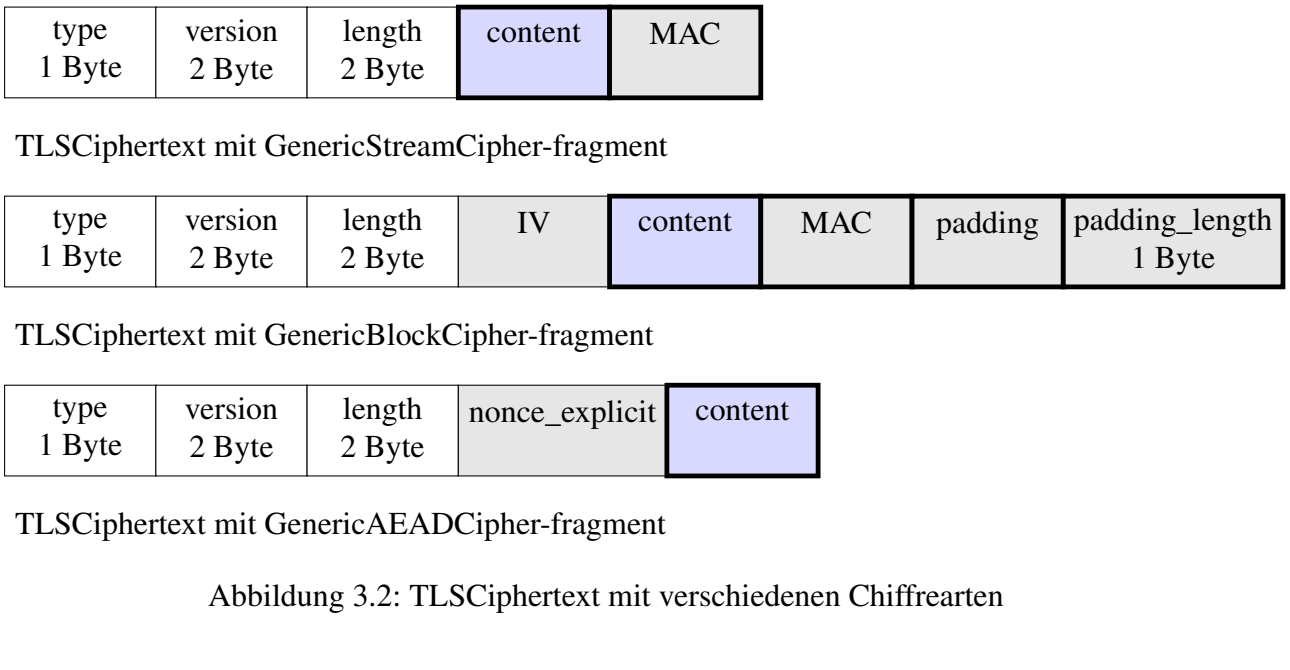

Abbildung 3.2: TLSCiphertext mit verschiedenen Chiffrearten

Bei der Nutzung von AEAD-Chiffren werden bei der Verschlüsselung zusätzlich zum Klartext und Schlüssel zwei zusätzliche Parameter verwendet: ein sogenanntes Nonce (eine einmalig verwendete, zufällige Eingabe) und zusätzliche Daten, in die die Sequenznummer der Nachricht, ihr Typ, ihre Version und ihre Länge einfließen (ähnlich zur Berechnung des MACs). Der explizite Teil des Nonce wird neben den verschlüsselten Daten übertragen. Der implizite Teil wird durch `server write IV` bzw. `client write IV` gebildet (vgl. Abschnitt 3.3). Die Notwendigkeit einer MAC-Berechnung entfällt bei diesem Verfahren.

3.3 Berechnung des Schlüsselmaterials

Bei der Berechnung von Schlüsseln verwendet TLS eine eigene Konstruktion einer Pseudo Random Function (PRF), die standardmäßig für alle Cipher-Suites verwendet wird und auf dem HMAC-Verfahren aufbaut:

```
PRF(secret, label, seed) = P_hash(secret, label + seed)

P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...
            mit A(0) = seed
                 A(i) = HMAC_hash(secret, A(i-1))
```

Aus `secret`, `label` und `seed` können so beliebige Mengen von pseudozufälligen Bits generiert werden (sieht man einmal von der Periode der Funktion ab).

Nach der ClientKeyExchange-Nachricht (siehe Abschnitt 3.4.9) sind Client und Server im Besitz des PreMasterSecret. Aus diesem und den in den Hello-Nachrichten übertragenen Zufallswerten (siehe Abschnitte 3.4.2 und 3.4.3) wird nun auf beiden Seiten das MasterSecret folgendermaßen generiert:

```
master_secret = PRF(pre_master_secret,
                    "master secret",
                    ClientHello.random + ServerHello.random)[0..47];
```

Aus diesem MasterSecret werden je nach verwendeten kryptographischen Verfahren Schlüssel für die Erstellung des MACs, für die Verschlüsselung zwischen Client und Server und für den impliziten Teil des Nonce bei AEAD-Chiffren berechnet:

```
client write MAC key
server write MAC key
client write encryption key
server write encryption key
client write IV
server write IV
```

Dazu werden solange Schlüsselblöcke nach dem folgenden Verfahren erstellt, bis genug Daten vorhanden sind, um alle benötigten Schlüssel konstruieren zu können:

```
key_block = PRF(SecurityParameters.master_secret,
                "key expansion",
                SecurityParameters.server_random +
                SecurityParameters.client_random);
```

3.4 Der TLS-Handshake

Das Handshake-Protokoll dient zur Herstellung einer gesicherten Verbindung. Hierbei werden verwendete TLS-Version und kryptographische Verfahren zwischen den Kommunikationspartnern vereinbart, optional ihre Identitäten authentifiziert und ein gemeinsames Geheimnis (das sogenannte PreMasterSecret) für die bereits beschriebene Generierung der während der eigentlichen Kommunikation verwendeten Schlüssel übertragen oder berechnet.

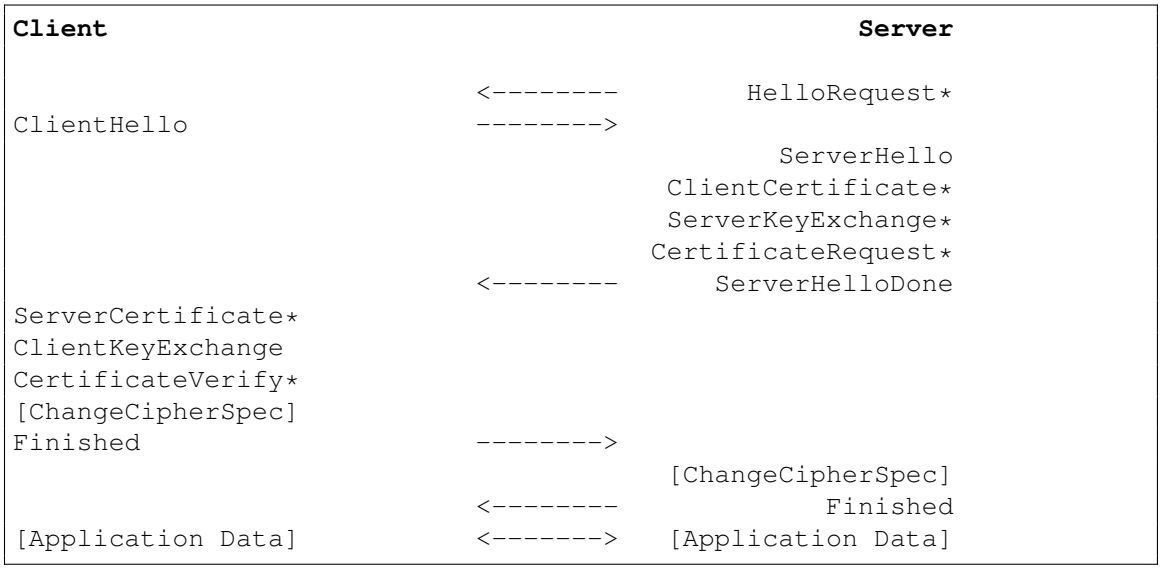

Abbildung 3.3: Nachrichtenverlauf beim vollständigen TLS-Handshake. Entnommen aus [DR08] und angepasst.

Eine Übersicht über die während eines vollständigen Handshakes ausgetauschten Nachrichten bietet Abbildung 3.3. Nachrichten, die - je nach gewünschten Eigenschaften der Verbindung - optional gesendet werden können, sind mit einem Stern (*) gekennzeichnet. Da die ChangeCipherSpec- und ApplicationData-Nachrichten einem eigenen Teilprotokoll entstammen, sind sie mit eckigen Klammern gekennzeichnet. Im Folgenden werden Aufbau der Handshake-Nachrichten und ihre Bedeutung nun im Detail betrachtet.

3.4.1 HelloRequest*

```
struct { } HelloRequest;
```

Diese Nachricht kann vom Server gesendet werden, wenn während einer bestehenden Verbindung ein neuer Handshake gewünscht wird. Dies kann beispielsweise dazu dienen Schlüssel für eine länger bestehende TLS-Verbindung nach einer gewissen Zeit neu auszuhandeln.

3.4.2 ClientHello

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
      case false:
          struct {};
      case true:
          Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

Mit der ClientHello-Nachricht initiiert der Client einen Verbindungsaufbau und sendet von ihm unterstützte Verfahren, die in der Verbindung genutzt werden können, sowie eine Identifikationsnummer für die Sitzung und einen Zufallswert.

Hierbei enthält `client_version` die neueste vom Client unterstützte TLS-/SSL-Version. `random` besteht aus einem 4-Byte großen Zeitstempel (UNIX-Format) und 28 zufälligen Bytes. Die `session_id` dient zur Identifikation einer Sitzung. Sie ist bei dem ersten Handshake leer und kann später dazu verwendet werden, bestehende Sitzungen wieder aufzunehmen (vgl. Abschnitt 3.5). Die Cipher-Suite-Liste enthält alle vom Client unterstützten Cipher-Suites geordnet nach Präferenz. Ebenso wird eine Liste von unterstützten Kompressionsalgorithmen übertragen. Optional kann auch eine Liste von gewünschten TLS-Extensions angegeben werden (vgl. Abschnitt 3.10).

3.4.3 ServerHello

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    select (extensions_present) {
```

```
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ServerHello;
```

In der ServerHello-Nachricht teilt der Server dem Client die (aus den in der ClientHello-Nachricht übertragenen) ausgewählten Verfahren mit und sendet die Identifikationsnummer sowie seinen Zufallswert.

In `server_version` steckt die höchste Version, die Server und Client unterstützen und die damit für die Kommunikation verwendet wird. `random` besteht äquivalent zur ClientHello-Nachricht aus einem 4-Byte Zeitstempel und 28 zufälligen Bytes, die vom Server berechnet wurden. Die `session_id` enthält entweder eine neu generierte Identifikationsnummer, die Identifikationsnummer einer wieder aufgenommen Sitzung oder kann auch leer sein, um anzugeben, dass die Sitzung nicht wieder aufgenommen werden kann. In `cipher_suite` und `compression_method` überträgt der Server die von ihm aus den vom Client übertragenen Listen ausgewählte Cipher-Suite bzw. den Kompressionsalgorithmus. In der Extensionliste gibt der Server alle vom Client gewünschten Extensions an, die er unterstützt.

3.4.4 ServerCertificate*

```
struct {
    ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

In dieser Nachricht sendet der Server seine Zertifikatskette zur Überprüfung seiner Identität. Das erste Zertifikat in der Liste bildet das Serverzertifikat, folgende Zertifikate müssen das jeweils vorhergehende zertifizieren. Der im Zertifikat enthaltene öffentliche Schlüssel muss zum ausgehandelten Schlüsselaustausch-Algorithmus passen. Wenn nicht anders vereinbart, wird für die Zertifikate das X.509v3-Format[4] verwendet.

3.4.5 ServerKeyExchange*

```
struct {
    select (KeyExchangeAlgorithm) {
      case dh_anon:
          ServerDHParams params;
      case dhe_dss:
      case dhe_rsa:
          ServerDHParams params;
          digitally-signed struct {
              opaque client_random[32];
              opaque server_random[32];
              ServerDHParams params;
          } signed_params;
      case rsa:
      case dh_dss:
      case dh_rsa:
          struct {} ; /* message is omitted for rsa, dh_dss, and
             dh_rsa */
    };
```

4. Ein Standard für Public-Key-Infrastrukturen, spezifiziert in RFC 5280.

```
} ServerKeyExchange;
```

Diese Nachricht wird nur für bestimmte Schlüsselaustausch-Verfahren gesendet, wenn die ServerCertificate-Nachricht nicht genügend Informationen zum Austausch des PreMasterSecret bietet (vgl. ClientKeyExchange-Nachricht).

`ServerDHParams` enthält dabei die öffentlichen Diffie-Hellman-Parameter p (die prime Ordnung der gewählten Gruppe), g (einen Erzeuger der Gruppe) und den öffentlichen Schlüssel $Y_s = g^{X_s} \mod p$, wobei X_s für den geheimen Schlüssel des Servers steht.

Im Fall eines nicht anonymen Diffie-Hellman-Schlüsselaustauschs werden diese Parameter mit gewähltem asymmetrischen Verfahren (DSS oder RSA) und dem zum öffentlichen Schlüssel aus der ServerCertificate-Nachricht gehörigen geheimen Schlüssel signiert.

3.4.6 CertificateRequest*

```
struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    SignatureAndHashAlgorithm supported_signature_algorithms<2^16-1>;
    DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

TLS unterstützt eine optionale Clientauthentifizierung. Mit dieser Nachricht kann der Client vom Server aufgefordert werden ebenfalls ein Zertifikat zu senden. Die `certificate_types`-Liste enthält alle Zertifikatarten (z.B. Zertifikat mit RSA-Schlüssel, ...), die vom Server unterstützt werden, und `supported_signature_algorithms` die unterstützten Signaturalgorithmen. In `certificate_authorities` kann eine Liste von erwarteten CAs übertragen werden.

3.4.7 ServerHelloDone

```
struct { } ServerHelloDone;
```

Mit dieser Nachricht signalisiert der Server das Ende des ServerHello und zugehöriger Nachrichten.

3.4.8 ClientCertificate*

Wenn von dem Server eine Clientauthentifizierung gefordert wurde, kann der Client in dieser Nachricht seine Zertifikatskette senden. Das Format entspricht dem der ServerCertificate-Nachricht.

3.4.9 ClientKeyExchange

```
struct {
    select (KeyExchangeAlgorithm) {
      case rsa:
          EncryptedPreMasterSecret;
      case dhe_dss:
      case dhe_rsa:
      case dh_dss:
      case dh_rsa:
      case dh_anon:
```

```
            ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

Wenn RSA als Schlüsselaustausch-Algorithmus vereinbart wurde, so wird das vom Client generierte PreMasterSecret mit dem (aus dem Serverzertifikat stammenden) öffentlichen Schlüssel des Servers verschlüsselt und gesendet. Es besteht aus der größten vom Client unterstützten Protokollversion (2 Bytes), um Version-Rollback-Angriffe zu verhindern (siehe Abschnitt 4.1), und 46 zufällig generierten Bytes.

```
struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;
```

Wenn der Schlüsselaustausch per Diffie-Hellman-Verfahren geschieht und der öffentliche Schlüssel des Clients nicht in seinem optional in der ClientCertificate-Nachricht gesendeten Zertifikat enthalten ist, sendet er in dieser Nachricht seinen öffentlichen DH-Schlüssel $Y_c = g^{X_c} \mod p$.

```
struct {
    select (PublicValueEncoding) {
      case implicit: struct { };
      case explicit: opaque dh_Yc<1..2^16-1>;
    } dh_public;
} ClientDiffieHellmanPublic;
```

Das PreMasterSecret wird dann als $Z = (Y_c)^{X_s} \mod p$ auf der Serverseite bzw. $Z = (Y_s)^{X_c} \mod p$ auf der Clientseite berechnet.

3.4.10 CertificateVerify*

```
struct {
    digitally-signed struct {
       opaque handshake_messages[handshake_messages_length];
    }
} CertificateVerify;
```

Diese Nachricht wird gesendet, falls ein Clientzertifikat vom Server angefordert wurde. Sie besteht aus einem mit dem geheimen Schlüssel des Clients signierten Hash der bisherigen Handshake-Nachrichten und dient zur Authentifikation des Clients.

3.4.11 ChangeCipherSpec

Diese Nachricht gehört zum ChangeCipherSpec-Protokoll (siehe Abschnitt 3.6). Auf diese Nachricht folgende Nachrichten werden mit den ausgehandelten Verfahren und Schlüsseln geschützt.

3.4.12 Finished

```
struct {
    opaque verify_data[verify_data_length];
} Finished;
```

Die Finished-Nachricht dient zur Verifikation von erfolgreichem Schlüsselaustausch und Authentifikation. Wie erwähnt ist dies die erste von den ausgehandelten Verfahren und Schlüsseln geschützte Nachricht. Daher kann hier überprüft werden, ob der Handshake erfolgreich verlaufen ist und beiden Kommunikationspartnern die gleichen Informationen vorliegen.

Die Nachricht besteht aus einem Hash über die bisher gesendeten bzw. empfangenen Nachrichten des Handshake-Protokolls zusammen mit dem MasterSecret:

```
verify_data = PRF(master_secret,
                  finished_label,
                  Hash(handshake_messages)) [0..verify_data_length-1];
```

Das `finished_label` wird durch „client finished“ auf der Client- bzw. „server finished“ auf der Serverseite gebildet. Der Hash wird durch die in der PRF verwendete Hashfunktion berechnet (vgl. Abschnitt 3.3). `verify_data_length` entspricht, wenn durch die Cipher-Suite nicht anders vorgegeben, 12 Bytes Länge.

Nachdem `verify_data` vom Server und Client jeweils mit dem für die Gegenseite berechneten Wert verglichen wurde, ist die Verbindung im Erfolgsfall aufgebaut.

Nun sind die `SecurityParameters` vereinbart und bilden zusammen mit den berechneten Schlüsseln (vgl. Abschnitt 3.3) den Verbindungszustand.

```
struct {
    ConnectionEnd           entity;
    PRFAlgorithm            prf_algorithm;
    BulkCipherAlgorithm     bulk_cipher_algorithm;
    CipherType              cipher_type;
    uint8                   enc_key_length;
    uint8                   block_length;
    uint8                   fixed_iv_length;
    uint8                   record_iv_length;
    MACAlgorithm            mac_algorithm;
    uint8                   mac_length;
    uint8                   mac_key_length;
    CompressionMethod       compression_algorithm;
    opaque                  master_secret[48];
    opaque                  client_random[32];
    opaque                  server_random[32];
} SecurityParameters;
```

Diese Informationen enthalten Angaben zu verwendeter Cipher-Suite, zum Kompressionsalgorithmus und das MasterSecret. Sie werden vom Record-Protokoll für die Verschlüsselung und Authentifizierung von Nachrichten verwendet.

3.5 Sitzungen, Verbindungen und der verkürzte Handshake

TLS erstellt beim ersten Handshake eine Sitzung zwischen Client und Server. Hierbei wird ein Sitzungsidentifikator gewählt, der beim ServerHello mitgesendet wird.

Ein Client kann nun später, wenn er den erhaltenen Sitzungsidentifikator in einer ClientHello-Nachricht mitschickt, eine alte Sitzung in Form einer neuen Verbindung wieder aufnehmen oder mehrere Verbindungen parallel aufbauen. Dabei werden die in den `SecurityParameters` hinterlegten Verfahren genutzt und aus dem ebenfalls hinterlegten MasterSecret sowie den in den Hello-Nachrichten übertragenen random-Werten neue Schlüssel berechnet (siehe Abschnitt 3.3).

Dadurch kommt der verkürzte Handshake mit weniger gesendeten Nachrichten aus, als ein neuer Handshake, wie in Abbildung 3.4 ersichtlich ist. So kann auf Neuberechnung des MasterSecret, Server- und Client-Validierung und Aushandlung der Cipher-Suite verzichtet werden. Durch die Finished-Nachricht können sich Client und Server durch das gleiche Schlüsselmaterial, das auf dem MasterSecret beruht, trotzdem sicher sein, mit dem optional authentifizierten Gegenüber zu kommunizieren.

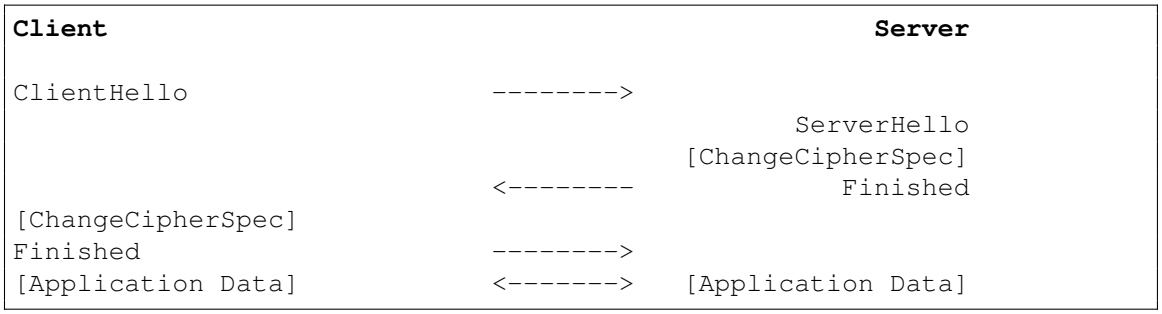

Abbildung 3.4: Nachrichtenverlauf beim abgekürzten TLS-Handshake. Entnommen aus [DR08].

3.6 ChangeCipherSpec-Protokoll

Das ChangeCipherSpec-Protokoll enthält lediglich eine Nachricht mit dem Wert 1. Das Empfangen dieser Nachricht signalisiert dem Empfänger, dass alle nachfolgend gesendeten Nachrichten mit den ausgehandelten kryptographischen Verfahren und Schlüsseln geschützt werden.
Gedanklich existieren bei der Ausführung des TLS-Handshakes vier Verbindungszustände: jeweils zwei Zustände für das Lesen und Schreiben, einer davon für den aktuellen Zustand (der sogenannte current read/write state) und einer für den Zustand, der ausgehandelte wird (der sogenannte pending read/write state). Während des Handshakes werden Verfahren und Schlüssel in den pending states gesetzt. Bei Empfang einer ChangeCipherSpec-Nachricht wird der pending read state in den current read state kopiert, sodass alle nachfolgenden empfangenen Nachrichten unter den neuen Schlüsseln und Verfahren verarbeitet werden. Entsprechend wird nach dem Senden der ChangeCipherSpec-Nachricht der pending write state in den current write state kopiert. Alle nun gesendeten Nachrichten werden nun ebenfalls durch ausgehandelte Verfahren geschützt.
Zu Beginn einer Verbindung ist in den current read/write states die Cipher-Suite TLS_NULL_WITH_NULL_NU
gesetzt (vgl. Abschnitt 3.9).

3.7 Alert-Protokoll

Das Alert-Protokoll dient dazu, Fehler zu versenden, die während der Kommunikation auftreten. Hierbei kann es sich zum Beispiel um fehlgeschlagene Überprüfung von entschlüsselten Nachrichten (bad_record_mac) oder fehlerhafte Zertifikatsüberprüfung (bad_certificate) handeln. Unterschieden wird zwischen Fehlern (fatal alert), die sofort zum Schließen der Sitzung führen, und Warnungen (warning alert). Eine Übersicht über alle Fehler findet sich in Abschnitt 7.2 von [DR08].

3.8 ApplicationData-Protokoll

Das ApplicationData-Protokoll ist zuständig für das Durchreichen von Anwendungsdaten, die von der Anwendungsschicht gesendet werden sollen. Die Daten werden durch das Record-Protokoll übertragen und damit durch die während des Handshakes ausgehandelten Verfahren geschützt.

3.9 Cipher-Suites

Eine Cipher-Suite legt fest, welche Algorithmen zum Schlüsselaustausch, zur Verschlüsselung und zur Berechnung des MACs verwendet werden und welche Eigenschaften (Schlüssellänge, Blocklänge, ...) diese besitzen. In der TLS 1.2-Spezifikation ([DR08]) sind 37 Cipher-Suites festgelegt.

Zum Schlüsselaustausch stehen `RSA` sowie verschiedene Varianten[5] des Diffie-Hellman-Verfahrens zur Verfügung. Zur Verschlüsselung sind die Stromchiffre `RC4_128` sowie die Blockchiffren `3DES_EDE_CBC`, `AES_128_CBC` und `AES_256_CBC` festgelegt. Zur Berechnung des MACs können `MD5`, `SHA1` und `SHA256` verwendet werden (jeweils wie erwähnt unter Nutzung von HMAC).

Laut Spezifikation muss zumindest `TLS_RSA_WITH_AES_128_CBC_SHA` von jeder konformen Implementation angeboten werden.

Einen Sonderfall bildet die Cipher-Suite `TLS_NULL_WITH_NULL_NULL`, die vor der Aushandlung der Cipher-Suite während des Handshakes als Standard festgelegt ist und weder Verschlüsselung noch MAC bietet.

Frühere Versionen (bis TLS 1.1) enthielten aufgrund von gesetzlichen Vorschriften zum Export von Kryptographie aus den USA Cipher-Suites, die durch Nutzung kürzerer Schlüssel leichter zu brechen sein sollten (so genannte export-geschwächte Cipher-Suites). Diese teilweise noch unterstützten Verfahren ermöglichen noch heute Angriffe auf SSL/TLS, z.B. den FREAK-Angriff in Abschnitt 4.11.

3.10 TLS-Extensions

TLS-Extensions werden dazu genutzt, das Protokoll um zusätzliche Funktionalität zu erweitern. Das Konzept wurde parallel zu TLS entwickelt und mit TLS 1.2 in den Standard aufgenommen.

In der ClientHello- und ServerHello-Nachricht können sich die Kommunikationspartner auf Extensions einigen, die von beiden Seiten unterstützt werden und im Verlauf der Sitzung genutzt werden können. Jeder Extension-Eintrag wird dabei durch ihren Typ und Extension-spezifische Daten gebildet.

5. `DH`: Zertifikat mit festen Diffie-Hellman-Parametern
`DHE`: Temporäre Generierung von DH-Parametern für jede Sitzung
`DH_anon`: Nicht-authentifizierte DH-Parameter

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;
```

Beispiele für solche Extensions sind Server Name Indication, die es einem Server erlaubt abhängig vom geforderten Host verschiedene Zertifikate auszuliefern, oder Encrypt-then-MAC, die es erlaubt, die Reihenfolge von Verschlüsselung und Authentifizierung einer Nachricht im Record-Protokoll zu tauschen. Eine aktuelle Liste registrierter TLS-Extensions wird durch die IANA (Internet Assigned Numbers Authority) bereitgestellt[6].

3.11 Frühere SSL-/TLS-Versionen und TLS 1.3

Im Folgenden soll kurz auf frühere Versionen von TLS/SSL und die entscheidenden Unterschiede zwischen diesen Versionen eingegangen sowie ein Überblick über Änderungen in dem noch nicht veröffentlichten TLS 1.3 gegeben werden.

SSL 1.0 wurde nie veröffentlicht.

SSL 2.0 war die erste Version, die öffentlich gemacht und auch patentiert wurde. In dieser Version bestanden einige große Schwachstellen: Der Handshake wurde noch nicht authentifiziert, so dass Angreifer beispielsweise die Cipher-Suite-Liste unbemerkt verändern konnten, viele schwache kryptographische Algorithmen wurden unterstützt und für Verschlüsselung und MAC-Berechnung wurden die gleichen Schlüssel verwendet. Auf einige dieser Schwachstellen wird genauer in Kapitel 4 eingegangen. Da die Spezifikation nicht veröffentlicht wurde, stammen diese Informationen aus [Mey14]. Die Unterstützung von SSL 2.0 wird für TLS-Implementierungen durch RFC 6176 ([TP11]) verboten.

In **SSL 3.0** wurden verschiedene Schlüssel zur Verschlüsselung und MAC-Berechnung eingeführt und die während der MAC-Berechnung genutzte Hashfunktion konfigurierbar gemacht (aber noch kein HMAC verwendet). Der Handshake wurde authentifiziert (durch den Inhalt der Finished-Nachricht) und in die Generierung des PreMasterSecret floss die Versionsnummer ein, um Version-Rollback-Angriffe zu verhindern. Außerdem wurden neue kryptographische Algorithmen eingeführt und weitere kleine Änderungen vorgenommen (vgl. [FKK11]). Von der Abwärtskompabilität von TLS-Implementierungen zu SSL 3.0 wird in RFC 7568 ([BTPL15]) abgeraten.

TLS 1.0 ist großenteils äquivalent zu SSL 3.0. Es wurde die Pseudo Random Function eingeführt, die allerdings noch anders spezifiziert war als in der aktuellen Version. Die Berechnung des MACs erfolgte nun durch eine HMAC-Konstruktion. Außerdem wurde eine ChangeCipherSpec-Nachricht vor der Finished-Nachricht vorgeschrieben, um den DropChangeCipherSpec-Angriff zu verhindern (vgl. [DA99]).

In **TLS 1.1** wurden explizite Initialisierungsvektoren für Blockchiffren im CBC-Modus vorgeschrieben, um den in Abschnitt 4.7 vorgestellten Angriff zu verhindern. Das Verhalten bei Padding-Fehlern wurde vorgeschrieben, um den Bleichenbacher-Angriff zu verhindern. Außerdem wurden die exportgeschwächten Cipher-Suites aus der Spezifikation entfernt (vgl. [DR06]).

6. http://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml

In **TLS 1.2** wurde der Gebrauch von SHA1 und MD5 in der Pseudo Random Function durch eine Cipher-Suite-abhängige Hashfunktion ersetzt. Zusätzlich wurde die Unterstützung von AEAD-Cipher-Suites eingefügt. TLS-Extensions und der Gebrauch von AES als Blockchiffre wurden ergänzt. Außerdem wurden DES und IDEA als Blockchiffren aus der Spezifikation entfernt, die Unterstützung von SSL 2.0 nicht mehr empfohlen und weitere kleine Änderungen vorgenommen (vgl. [DR08]).

TLS 1.3 liegt momentan als Draft vor ([Res15], Version von Oktober 2015]). Die hier dargestellten Informationen stellen also lediglich den aktuellen Entwicklungsstand dar und sind daher nicht als endgültig und nur mit Bedacht zu betrachten.
In dem aktuellen Draft wird Elliptic Curve Cryptography für das Diffie-Hellman-Verfahren hinzugefügt, die Unterstützung für alle SSL-Versionen und auch Kompression komplett entfernt (wahrscheinlich als Maßnahme gegen den Angriff aus Abschnitt 4.9). Die größte Veränderung ist die Ausmusterung von Strom- und Blockchiffren im CBC-Modus zur Verschlüsselung. Es werden nur noch AEAD-Chiffren unterstützt, womit auch die separate Berechnung eines MACs entfällt. Außerdem wurde die PRF-Konstruktion durch die Verwendung von HKDF (HMAC-based Key Derivation Function, siehe [Kra10]) ersetzt. Zusätzlich wird der ContentType eines TLS-Record nur noch verschlüsselt übertragen und Records können zum Verstecken der Nachrichtenlänge mit Padding beliebiger Länge (unter Berücksichtigung der max. Record-Länge) versehen werden. Weitere Änderungen befassen sich mit Möglichkeiten zu schnellerem Senden von ApplicationData-Nachrichten während des Handshakes.

https://www.trustworthpulse/ - SSL-Versionsverbreitung ergänzen?

TLS 1.0 RFC 2246 - http://tools.ietf.org/html/rfc2246

TLS 1.1 RFC 4346 - http://tools.ietf.org/html/rfc4346

TLS 1.2 RFC 5246 - http://tools.ietf.org/html/rfc5246

TLS 1.3 Draft -https://tools.ietf.org/html/draft-ietf-tls-tls13-07

TLS Extensions Z.B.
RFC 3546 - http://tools.ietf.org/html/rfc3546,
RFC 3466 - http://tools.ietf.org/html/rfc4366,
RFC 6066 - http://tools.ietf.org/html/rfc6066

3.12 Implementierungen

An dieser Stelle sollen auch noch bestehende Implementierungen von SSL/TLS erwähnt werden, auch wenn der Fokus der Arbeit auf der Protokollspezifikation selbst liegt.

Die meistgenutzte Implementierung, die unter anderem auch im häufig verwendeten Apache-Webserver zum Einsatz kommt, ist OpenSSL, eine Open-Source-Implementierung in C. In Produkten von Microsoft wird die Bibliothek SChannel, in Apple-Anwendungen Secure Transport und in Google Chrome und Mozilla-Produkten NSS verwendet. Auch einige Programmiersprachen bringen eigene Implementierungen mit. Ein Beispiel hierfür ist die Java Secure Socket Extension (JSSE).

Zusätzlich gibt es viele weitere seltener genutzte Implementierungen wie GnuTLS, PolarSSL, LibreSSL oder Amazon s2n, die vollständig neu entwickelt wurden oder als Forks bestehender Implementierungen entstanden sind.

Viele Angriffe auf TLS-gesicherte Verbindungen, die bekannt geworden sind, waren Angriffe auf Implementierungen und nicht auf die Protokollspezifikation selbst (auf diese zweite Art von Angiffen wird in Kapitel 4 eingegangen). Ein Beispiel ist der Heartbleed[7] getaufte Bug in OpenSSL, der es wegen eines Programmierfehlers ermöglichte, Speicherinhalte des Servers auszulesen. Auf solche Angriffe, die lediglich einzelne Implementierungen betreffen, soll im Rahmen dieser Arbeit nicht weiter eingegangen werden.

Eine gelungene Übersicht über bestehende Implementierungen, die tiefer ins Detail geht, ist in [Mey14] zu finden.

7. http://heartbleed.com/ Abgerufen am 13.8.2015

4 Angriffe gegen SSL und TLS

Mehr Visualisierungen im ganzen Kapitel, falls sich das anbietet

In diesem Kapitel wird auf entdeckte Angriffe gegen SSL/TLS eingegangen. Wie bereits erwähnt sollen hierbei Angriffe gegen die Spezifikation selbst und nicht gegen spezielle Implementierungen im Fokus stehen.

Eine gute Übersicht zu bisherigen Angriffen auf TLS findet sich in [MS13]. Viele Schwächen früherer Protokollversionen bis SSL 3 sind in [WS96] zu finden.

4.1 Version Rollback

Ein Angreifer kann eine SSL 3.0-konforme ClientHello-Nachricht durch Ändern des Versionsfeldes in der Nachricht so modifizieren, dass der Server eine SSL 2.0-Verbindung aufbaut. So kann der Angreifer alle Schwächen der älteren Protokollversion ausnutzen.

Durch die Finished-Nachricht ab SSL 3.0, die alle Handshake-Nachrichten authentifiziert, wird dieser Angriff verhindert.

Um Kompatibilität mit der älteren Version erhalten, aber Version-Rollback-Angriffe trotzdem zu erkennen, wurde in der SSL 3.0-Spezifikation [FKK11] vorgeschrieben, dass bestimmte Bytes des PKCS#1-Paddings (siehe [JK03]) einen festen Wert erhalten sollten, falls der Client SSL 3.0 unterstützt. Ein Schwachpunkt könnte laut [WS96] immer noch die Wiederaufnahme einer SSL 3.0-Sitzung durch eine SSL 2.0 ClientHello-Nachricht sein. Dieses sollte in Implementierungen verhindert werden.

Auch in neueren Browsern kann dieser Angriff noch zum Problem werden, falls Fallback-Lösungen auf SSL 2.0 implementiert sind, wenn ein Verbindungsversuch scheitert. Durch das Verbot von SSL 2.0-Unterstützung (vgl. Abschnitt 3.11) sollte dieser Angriff für TLS-Versionen keine Auswirkungen mehr haben.

4.2 Ciphersuite Rollback

Ein gegen SSL 2.0 bestehender Angriff ermöglichte aktiven Angreifern die während des Handshake-Protokolls übertragenen Listen von unterstützten Cipher Suites so zu verändern, dass schwache kryptographische Verfahren erzwungen werden konnten (oftmals exportgeschwächte Verfahren mit kürzeren Schlüsselängen).

Ab SSL 3.0 wird dieser Angriff dadurch verhindert, dass die Finished-Nachrichten von Client und Server jeweils einen mit dem MasterSecret berechneten MAC über die Nachrichten des Handshake-Protokolls enthalten, der die Integrität dieser Nachrichten bestätigt.

Eine detaillierte Übersicht ist in [WS96] zu finden.

4.3 Verhindern der ChangeCipherSpec-Nachricht

Für den Sonderfall einer SSL-Verbindung, die lediglich die Integrität der Nachrichten schützen soll, aber nicht verschlüsselt, lässt sich ausnutzen, dass der in der Finished-Nachricht gesendete MAC die ChangeCipherSpec-Nachricht nicht mit einschließt. Dadurch kann ein aktiver Angreifer diese Nachrichten abfangen und nicht weiterleiten, sodass die Verbindungspartner die Integritätsprüfung nicht einsetzen. Ein Angreifer ist so in der Lage, gesendete Nachrichten zu verändern.

Theoretisch ist der Angriff unter bestimmten Voraussetzungen und schwacher Kryptographie auch bei verschlüsselten Verbindungen möglich. Dazu muss die empfangene Finished-Nachricht vor dem Weiterleiten entschlüsselt werden. In bestimmten Fällen ist zwar genug bekannter Klartext vorhanden, um einen Brute-Force-Angriff auf den Schlüssel zu erlauben, aber selbst bei exportgeschwächten 40-Bit Schlüsseln ist der Rechenaufwand hierfür sehr groß und unter praktischen Gesichtspunkten kaum unbemerkt ausführbar.

Alle TLS-Versionen verhindern diesen Angriff dadurch, dass sie eine ChangeCipherSpec-Nachricht vor der Finished-Nachricht explizit vorschreiben.

Details zu diesem Angriff sind in [WS96] zu finden.

4.4 Bleichenbacher-Angriff

Daniel Bleichenbacher stellte 1998 in [Ble98] einen sogenannten Adaptive-Chosen-Ciphertext-Angriff gegen RSA-basierte Protokolle vor. Im Fall von SSL wird versucht, das PreMasterSecret, das während des Handshakes RSA-verschlüsselt gesendet werden kann, zu erhalten.

Der Angriff basiert auf dem festen Format nach PKCS#1 (siehe [JK03]) formatierter Nachrichten, wie in Abbildung 4.1 dargestellt. Die ersten beiden Bytes haben immer den gleichen Wert. Danach folgen das aus zufälligen Bytes ungleich null bestehende Padding und die Daten, getrennt durch ein Nullbyte. Nachrichten in diesem Format werden als Integer interpretiert, per RSA verschlüsselt und versendet. Der Empfänger entschlüsselt die Nachricht, überprüft das Format des als Bytekette interpretierten Ergebnisses und kann dann die Daten wieder extrahieren.

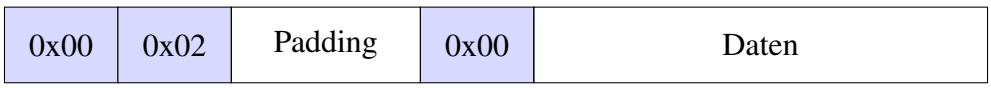

Abbildung 4.1: PCKS #1-Format

Voraussetzung für diesen Angriff ist der Zugriff auf ein Orakel, das dem Angreifer für eine verschlüsselte Nachricht lediglich mitteilt, ob das Padding der entschlüsselten Nachricht korrektes Format besitzt.

Im Folgenden sei (n,e) ein öffentlicher RSA-Schlüssel und (n,d) der zugehörige geheime Schlüssel. Der Angreifer möchte eine Nachricht $m \equiv c^d \mod n$ erhalten, für die er im Besitz von c ist.

Dazu wählt er eine Zahl s, berechnet $c' \equiv cs^e \mod n$ und sendet c' an das Orakel. Wenn das Orakel korrektes Format signalisiert, dann weiß der Angreifer, dass die ersten zwei Bytes von

$$(c')^d \equiv (cs^e)^d \equiv c^d s \equiv ms \mod n$$

$0x00$ und $0x02$ sind. Mit diesem Wissen lässt sich ein neuer Wert s wählen, der weitere Informationen über m enthüllt. Details zu diesem iterativen Verfahren sind in [Ble98] zu finden. Der Autor schätzt die Anzahl an nötigen Orakelanfragen mit etwa 2^{20} ab.
Das Orakel lässt sich auf zwei Weisen erhalten. Entweder gibt die Implementierung detaillierte Fehlermeldungen über ungültiges PKCS-Format zurück oder ermöglicht durch Zeitunterschiede bei der Verarbeitung gültiger und ungültiger Nachrichten einen Timing-Seitenkanal-Angriff.

In TLS ab Version 1.0 wird der Angriff dadurch verhindert, dass bei ungültigem PKCS-Format ein zufälliges PreMasterSecret erzeugt wird, mit dem der Handshake fortgesetzt wird. Dadurch scheitert der Handshake erst bei der Überprüfung der Finished-Nachricht und enthüllt keine Informationen über gültiges oder ungültiges Format.

4.5 Padding-Orakel-Angriff

In [Vau02] beschreibt der Autor einen Angriff zur Erlangung des Klartextes, bei dem das für Blockchiffren nötige Padding im CBC-Modus ausgenutzt wird. Durch das vorgegebene Format des Paddings und die Tatsache, dass das Padding bei TLS nicht durch den MAC geschützt ist (vgl. Abschnitt 3.2), ermöglicht es theoretisch in einer relativ kleinen Zahl von Anfragen die Berechnung des Klartextes.

Das verwendete Padding besteht immer aus Bytes mit dem Wert n, wobei n die nötige Anzahl an Paddingbytes bis zum Erreichen eines Vielfachens der Blocklänge bezeichnet. Das Padding kann also folgende Werte annehmen: $1, 22, 333, 4444, \ldots$. Voraussetzung für diesen Angriff ist der Zugriff auf ein Orakel, das dem Angreifer für eine verschlüsselte Nachricht lediglich mitteilt, ob das Padding der entschlüsselten Nachricht korrektes Format besitzt. Im Folgenden bezeichnet $C(x)$ die Verschlüsselung des Blockes x und $C^{-1}(y)$ die Entschlüsselung von y.
Wenn der Angreifer nun das letzte Byte eines Chiffretextblocks y erhalten möchte, so sendet er $r + y$ mit $r = r_1, \ldots, r_b$ als zufällige Bytes und b als Blocklänge (in Byte) an das Orakel. Bei der Entschlüsselung im CBC-Modus wird der letzte Chiffretextblock (hier y) entschlüsselt und mit dem vorletzten Block XOR-verknüpft, um den Klartextblock zu erhalten. Dieser Block (hier x) wird dann auf gültiges Padding überprüft:

$$x = C^{-1}(y) \oplus r$$

Wenn das Orakel gültiges Padding signalisiert, dann ist am wahrscheinlichsten, dass x auf 1 endet und somit das letzte Byte von $C^{-1}(y) = r_b \oplus 1$ ist. Bei ungültigem Padding wird ein neuer Wert r_b gewählt und das Orakel neu befragt.
In [Vau02] wird ein Algorithmus, mit dem auch die unwahrscheinlicheren Fälle von längerem Padding abgedeckt werden, und ein Verfahren, um aus dem letzten Byte einen kompletten Block zu erhalten, angegeben.

Praktisch konnte das Verfahren nicht eingesetzt werden, da SSL 3.0 für Paddingfehler und MAC-Überprüfung gleiche Fehlermeldungen (bad_record_mac) ausgibt. In TLS 1.0 und 1.1 gibt es getrennte Fehlermeldungen (bad_record_mac und decryption_failed), so dass der Angriff

theoretisch möglich wäre. Allerdings werden die Fehler über das Alert-Protokoll verschlüsselt gesendet, so dass ein Angreifer die Fehlerart anders erhalten muss (beispielsweise über Log-Einträge). TLS 1.2 verbietet aus diesem Grund das Senden von decryption_failed-Fehlern.
Ein weiterer Nachteil für den Angreifer ist, dass es sich bei bad_record_mac- und decryption_failed-Fehlern um fatal alerts handelt, die zum Abbruch der Sitzung führen.

In [CHVV03] beschreiben die Autoren eine Umsetzung des Angriffs auf TLS-gesicherte IMAP-Verbindungen zur Erlangung von Passwörtern. Hierbei wird das Problem ununterscheidbarer und verschlüsselter Fehlermeldungen durch einen Timing-Angriff umgangen. Außerdem bedenken die Autoren das Abbrechen der Sitzung durch Nutzung vieler paralleler Sitzungen mit dem gleichen verschlüsselten Aufruf, wie es bei der Authentifizierung im IMAP-Protokoll der Fall ist.

4.6 Lucky Thirteen

In [AFP13] stellen die Autoren weitere auf [Vau02] basierende Angriffe vor, die ebenfalls auf Timing-Angriffen zur Erkennung falschen Paddings und mehrere Verbindungen setzen.

4.7 Chosen-Plaintext-Angriff gegen bekannte IVs

In [Bar04] stellt der Autor einen Angriff vor, der die Art ausnutzt, wie die für den CBC-Modus nötigen Initialisierungsvektoren (IV) von TLS bereitgestellt wurden. Durch die Nutzung des letzten Ciphertextblocks der letzten Nachricht als IV der neuen Nachricht lässt sich unter bestimmten Voraussetzungen ein Chosen-Plaintext-Angriff durchführen. Der Autor beschreibt eine Möglichkeit unter Nutzung von Browser-Plugins über HTTPS übertragene Passwörter oder PINs herauszufinden. In [Bar06] verbessert der Autor seinen Angriff durch die Nutzung von Java-Applets anstelle von Browser-Plugins.

Der eigentliche Angriff entspricht dem folgenden Prinzip: Wenn eine Nachricht $C = C_0, \ldots, C_l$ gesendet wurde, wird für die nächste Nachricht C_l als IV verwendet werden. Wenn der Angreifer überprüfen möchte, ob ein Klartextblock $P^* = P_j$ zu C_j verschlüsselt wurde, so bringt er einen Sender dazu eine Nachricht P' mit dem ersten Block $P'_1 = C_{j-1} \oplus C_l \oplus P^*$ zu verschlüsseln und erhält als ersten Chiffretextblock:

$$\begin{aligned} C'_1 &= C_K(P'_1 \oplus \mathrm{IV}) \\ &= C_K(P'_1 \oplus C_l) \\ &= C_K(C_{j-1} \oplus C_l \oplus P^* \oplus C_l) \\ &= C_K(C_{j-1} \oplus P^*) \end{aligned}$$

Außerdem gilt auch $C_j = C_K(P_j \oplus C_{j-1})$. Der Angreifer kann nun überprüfen, ob $C'_1 = C_j$ und damit $P^* = P_j$ gilt, ob also seine Wahl für den gesuchten Klartextblock stimmt.

Seit TLS 1.1 werden explizite IV vorgeschrieben. Hierzu besteht jede verschlüsselte Nachricht aus einem Block mehr als Klartextblöcken. Dieser erste Block bildet den IV für die restliche Verschlüsselung. Da dieser IV nicht vor dem Empfang der Nachricht bekannt ist, wird der hier beschriebene Chosen-Plaintext-Angriff verhindert.

4.8 BEAST

In [DR11] und in einem Konferenzbeitrag auf der ekoparty Security Conference 2011 wurde von den Autoren das Tool BEAST vorgestellt, das die Ideen aus [Bar04] aufgreift. Die Autoren erweiterten den Angriff jedoch auf einen sogenannten block-wise chosen-boundary Angriff, bei dem der Angreifer die Lage der Nachricht in den verschlüsselten Blöcken verändern kann. Die Autoren zeigten auch die praktische Umsetzbarkeit am Beispiel des Entschlüsselns einer über HTTPS gesendeten Session-ID.

4.9 CRIME

Auf der ekoparty Security Conference 2012 stellten die Entdecker des BEAST-Angriff einen weiteren Angriff vor, der die (optionale) Kompression in TLS nutzt, um beispielsweise Cookiedaten zu stehlen.

Dabei wird ausgenutzt, dass Kompressionsalgorithmen bereits verwendete Zeichenketten beim erneuten Auftreten verkürzen. Wird nun beispielsweise ein HTTP-Header wie `Cookie: twid= secret` gesendet, kann ein Angreifer durch das Einbringen von `Cookie: twid= a...` und `Cookie: twid= s...` einen Längenunterschied der Nachrichten feststellen und so das Geheimnis zeichenweise erhalten.

Als Folge wurde Kompressionsunterstützung in Firefox und Chrome deaktiviert. Kompression ist im aktuellen Draft von TLS 1.3 nicht mehr enthalten [Res15].

4.10 Poodle

In [MDK14] nutzen die Autoren den erneuten Verbindungsversuch mit älteren Protokollversionen, wenn der Handshake fehlschlägt (SSL 3.0-Fallback), der in vielen TLS-Implementierungen eingesetzt wird. Darauf aufbauend beschreiben sie einen Angriff, der bestehende Schwächen in der RC4-Chiffre bzw. in der Nicht-Prüfung von Padding im CBC-Modus in SSL 3.0 ausnutzt, um Cookiedaten zu stehlen. Von der Unterstützung von SSL 3.0 wird in [BTPL15] abgeraten.

RC4 noch irgendwo unterbringen? Vllt bei den Ciphersuites? http://www.isg.rhul.ac.

4.11 FREAK

Eine Gruppe von Wissenschaftlern entdeckte eine Möglichkeit, wie ein Angreifer die Kommunikationspartner während des Handshakes zur Nutzung schwacher Kryptographie (RSA export cipher suite) bringen kann. Weiterhin zeigten sie in [BDLF$^+$] die Machbarkeit der Faktorisierung der entsprechenden RSA-Module und die Praxistauglichkeit des Angriffs.

Der Angriff beruht auf Fehlern in TLS-Implementierungen, die schwache RSA-Schlüssel vom Server akzeptieren, selbst wenn sie die Cipher-Suites nicht anbieten.

4.12 logjam

In [ABD+] beschreiben die Autoren mehrere Angriffe gegen die Nutzung von Diffie-Hellman-Schlüsselaustausch während des TLS Handshakes. Ein Angriff richtet sich gegen kleine DH-Parameter (DHE-EXPORT), ein weiterer nutzt die weite Verbreitung von standardisierten DH-Parametern, um mittels Vorberechnung bestimmter Werte schneller diskrete Logarithmen für beim DH-Verfahren gesendete Nachrichten zu berechnen.

4.13 Zertifikate und Verwandtes

Viele Probleme, die in den letzten Jahren aufgetreten sind, betreffen nicht das TLS-Protokoll direkt, sondern die Erstellung und Validierung von (insbesondere) Server-Zertifikaten, und seien deshalb nur am Rande erwähnt. Ein guter Überblick ist in [MS13] zu finden.

Oftmals richteten sich diese Angriffe gegen mangelnde Zertifikatvalidierung in TLS-Implementierungen. Fehler, die hier gemacht wurden, beinhalten unter anderem komplett fehlende Validierung, keine Überprüfung des Zertifikatsbesitzers oder die Akzeptanz unsignierter oder abgelaufener Zertifikate.

Auch fehlende Sorgfalt bei der Zertifikatserstellung durch Certificate Authorities wurde schon von Angreifern ausgenutzt. Hierbei waren beispielsweise die fehlerhafte Validierung von übermittelten Servernamen, die fehlerhafte Ausgabe von intermediate-Zertifikaten[1] und mangelhaft abgesicherte Server oder Benutzerkonten, die es Angreifern erlaubten, signierte Zertifikate auszustellen, Gründe für erfolgreiche Angriffe.

Der Vollständigkeit halber sei hier auch noch die notwendige Sicherheit des privaten Serverschlüssels hingewiesen. Gelangt ein Angreifer in seinen Besitz, so kann er den Datenverkehr problemlos mitlesen oder verändern.

1. Intermediate-Zertifikate sind Zertifikate, die es erlauben, andere Zertifikate zu signieren.

5 TLS in der Lehre

Im Folgenden soll darauf eingegangen werden, warum sich die Beschäftigung mit TLS in der Hochschullehre im Bereich der IT-Sicherheit anbietet.

In Abschnitt 3 wurde bereits die Bedeutung und umfassende Verwendung von TLS erwähnt. Dies ist ein Grund dafür, dass jeder, der sich detaillierter mit IT-Sicherheit beschäftigt, auch die Grundlagen von TLS verstehen sollte. Auch viele Angriffe, die gegen TLS entwickelt wurden, und die entsprechenden Gegenmaßnahmen bieten gute Möglichkeiten, um in der Lehre eingesetzt zu werden. Je nach gewünschten Schwerpunkten lassen sich beispielsweise Angriffe im Bereich der Kryptographie oder Protokollspezifikation nutzen. Sogar eine Betrachtung des gesellschaftlichen bzw. politischen Kontexts bietet sich an[1]. Zusätzlich spricht für einen Einsatz in der Lehre auch die ebenfalls bereits erwähnte einfache Protokollspezifikation, durch die der Blick auf wesentliche Prinzipien der IT-Sicherheit erleichtert wird. Auf einige dieser Prinzipien soll im nächsten Abschnitt eingegangen werden.

5.1 Schwerpunkte für die Lehre

In [SS11] wird das didaktische Prinzip nach J.S. Brunner erwähnt, wonach die Lehre sich „in erster Linie an den Strukturen der zugrundeliegenden Wissenschaft orientieren soll“. Diese Strukturen werden auch Fundamentale Ideen genannt.

Hierbei handelt es sich um „langlebige Konzepte“, die die „Übertragung (Transfer) früher erworbener Kenntnisse auf [neue] Situationen“ [SS11] ermöglichen sollen . Insbesondere der nichtspezifische Transfer sollte den Autoren zufolge an allgemeinbildenden Schulen und in der Hochschullehre im Vordergrund stehen. Dieser beschreibt das Lernen von grundlegenden Begriffen, Prinzipien und Denkweisen - im Gegensatz zum spezifischen Transfer, dem geringfügigen Anpassen einer bekannten Situation an ein ähnliches Problem.

Kennzeichnend für Fundamentale Ideen sind unter anderem das Horizontal- und Zeitkriterium, also die umfassende Anwendbarkeit oder Erkennbarkeit in vielen Bereichen einer Wissenschaft und ebenso die längerfristige Relevanz der Idee.

Als Empfehlung für die Hochschuldidaktik fassen die Autoren dieses Prinzip folgendermaßen zusammen:

> „Jeder Student wird im Laufe seines Berufslebens vermutlich mehreren Paradigmenwechseln der Informatik gegenüberstehen, wobei jeweils ein größerer Teil seines Wissens überflüssig oder fehlerhaft wird. Daher sollten die Fähigkeiten, die er während des Studiums erwirbt, möglichst robust gegenüber neuen wissenschaftlichen Entwicklungen sein und ihn befähigen, Paradigmenwechsel zu bewältigen. Folglich müssen Studenten ein Bild von den dauerhaften Grundlagen, den Fundamentalen

1. Die Vorschrift für die Bereitstellung von exportgeschwächten Cipher-Suites durch die US-Regierung könnte hier beispielsweise ein Ausgangspunkt sein.

> Ideen, Prinzipien, Methoden und Denkweisen der Informatik erlangen. Dazu sind in Vorlesungen und Lehrbüchern stets die Fundamentalen Ideen, die sich hinter den jeweils behandelten Sachgebieten verbergen, herauszuarbeiten, zu betonen, zu anderen Teilgebieten in Beziehung zu setzen und so in einen übergeordneten Zusammenhang einzuordnen.“ [SS11]

Auch wenn das Prinzip Fundamentaler Ideen umfassender ist als das eingeschränkte Thema dieser Arbeit, so lässt sich doch eine klare Empfehlung für die Nutzung von TLS (oder anderer konkreter Protokolle oder Verfahren) in der Hochschullehre ableiten. Es sollten insbesondere Prinzipien betrachtet werden, die als häufig verwendet und auch langfristig gültig für den Bereich der IT-Sicherheit, der Kryptographie oder der Sicherheitsprotokolle angesehen werden können. Dieses Vorgehen wird auch von Klüver und Klüver als zentrale Aufgabe der Didaktik ausgemacht und als didaktische Reduktion, also „die Rückführung komplexer Sachverhalte auf ihre wesentlichen Elemente, um sie für Lernende überschaubar und begreiflich zu machen“ [KK12], bezeichnet.

Im Folgenden sollen einige Vorschläge für solche Prinzipien aufgeführt werden, deren Bedeutung durch Angabe weiterer Bereiche, in denen sie zur Anwendung kommen, bestätigt wird.

5.1.1 Hybride Kryptosysteme

In Kapitel 2 wurde bereits auf das Problem des Schlüsselaustausches bei symmetrischen Algorithmen zur Ver- und Entschlüsselung eingegangen. Durch asymmetrische Verfahren lässt sich dieses Problem leicht lösen. Diese benötigen jedoch deutlich mehr Zeit als symmetrische Verfahren für die Verarbeitung von Daten.

Um die Vorteile beider Lösungen zu kombinieren, werden häufig hybride Kryptosysteme eingesetzt: Es wird für jede neue Kommunikation ein symmetrischer Schlüssel zufällig generiert (daher Sitzungsschlüssel genannt) und dem Kommunikationspartner mit dessen öffentlichen Schlüssel verschlüsselt zugesendet. Eine andere Möglichkeit ist die Schlüsselvereinbarung per Diffie-Hellman-Verfahren (vgl. Abschnitt 2.1.3). Nach der Entschlüsselung oder Vereinbarung sind beide Partner im Besitz des gleichen Sitzungsschlüssels und können ihre Kommunikation symmetrisch verschlüsseln [Sch06].

Dieses Prinzip kommt neben seiner Anwendung in TLS unter anderem auch in IPSec und PGP zum Einsatz.

5.1.2 Problem des Schlüsselaustausches und Authentifizierung

Wie ebenfalls bereits in Kapitel 2 erwähnt, besteht bei der Nutzung von asymmetrischen Verfahren jedoch das Problem, die Identität des Besitzers des öffentlichen Schlüssels sicherzustellen.

Dieses Problem soll kurz erläutert werden. Möchte eine Person A eine Nachricht asymmetrisch verschlüsselt mit B austauschen, so benötigt sie den öffentlichen Schlüssel K^B_{public} von B. Ein Angreifer M, der die Kommunikation zwischen A und B lesen und verändern kann (ein sogenannter Man-in-the-Middle), kann diesen öffentlichen Schlüssel bei der Übermittlung durch seinen eigenen K^M_{public} austauschen. Die Nachricht von A kann er nun mit seinem geheimen Schlüssel entschlüsseln. Damit sein Angriff unbemerkt bleibt, kann er die Nachricht anschließend noch mit K^B_{public} verschlüsseln und an B senden. Dieser Angriff ist in Abbildung 5.1 dargestellt.

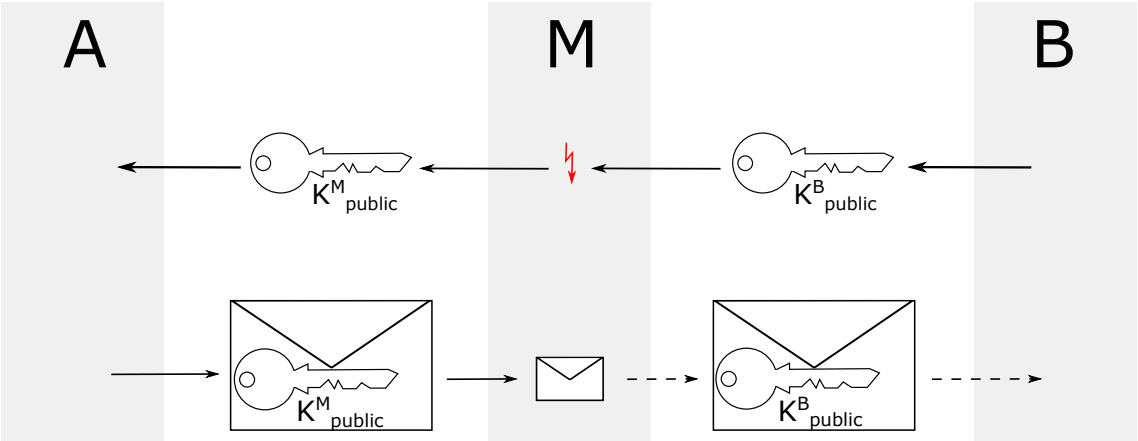

Abbildung 5.1: Man-in-the-middle-Angriff auf den Schlüsselaustausch

Um diesen Angriff zu verhindern, muss es eine Möglichkeit geben, den Besitzer des Schlüssels zu verifizieren. Für diese Aufgabe wird in SSL/TLS auf Zertifikate und eine Public-Key-Infrastruktur (PKI) gesetzt. Ein Zertifikat enthält den öffentlichen Schlüssel und die Informationen des Besitzers bestehend aus dem Namen oder der URL des Servers und weiteren Angaben. Dieses Zertifikat wird von einer vertrauenswürdigen Instanz, der Certificate Authority (CA), nach der Überprüfung der Besitzeridentität mit ihrem geheimen Schlüssel signiert. Ein Empfänger des Zertifikats kann nun das Zertifikat mit dem öffentlichen Schlüssel der CA überprüfen, in dessen Besitz er im Vorwege sein muss[2]. Danach ist der Besitzer und der zugehörige öffentliche Schlüssel verifiziert und kann zum Verschlüsseln von Nachrichten genutzt werden[3].

Die Nutzung von Zertifikaten und PKIs findet unter anderem auch in IPSec oder S/MIME Verwendung.

5.1.3 Seitenkanalangriffe

Seitenkanalangriffe sind Angriffe, die nicht das kryptographische Verfahren direkt angreifen, sondern versuchen Informationen über die Nachricht oder verwendete Schlüssel aus anderen Kanälen zu erhalten. Hierbei kann es sich beispielsweise um Zeit- oder Stromverbrauchsmessung handeln, aber auch um Messung von elektromagnetischer Abstrahlung, Schall oder sogar des Erdungspotentials eines Rechners.

Wenn sich unterschiedliche Schlüssel- oder Nachrichtenbits verschieden auf die Werte des Seitenkanals auswirken, so lassen sich Schlüssel bzw. Nachricht oftmals aus den gemessenen Informationen ableiten. Gerade bei einfachen Geräten wie Smartcards können schon einfache Angriffe erfolgreich sein.

Um vor Seitenkanalangriffen geschützt zu sein, muss eine Implementierung sich für unterschiedliche Eingaben möglichst gleich verhalten. Ein Beispiel als Maßnahme gegen Timing-Angriffe (also die Messung der benötigten Zeit einer Operation) ist die Einführung einer konstanten Zeit für die Ausführungsdauer. Dazu wird eine Zeit d gewählt, die länger ist als jede mögliche

2. Heutige Browser und Betriebssysteme werden bereits mit Listen solcher CAs ausgeliefert.
3. Die Schwächen, die in diesem System bestehen können, sind vielfältig, liegen jedoch nicht im Fokus dieser Arbeit. Erste Hinweise sind in Abschnitt 4.13 und in [FSK10] zu finden.

benötigte Ausführungsdauer t der Operation. Nach der Ausführung wird immer die Zeit $d - t$ gewartet, sodass die Operation insgesamt die konstante Zeit d benötigt. Diese Möglichkeit schützt natürlich nicht vor anderen Seitenkanalangriffen, für deren Abwehr andere Schutzmaßnahmen getroffen werden müssen [FSK10].

Seitenkanalangriffe in Form von Timing-Angriffen werden auch bei einigen Angriffen gegen SSL/TLS eingesetzt, wie beispielsweise bei dem Bleichenbacher-Angriff (Abschnitt 4.4) oder dem Padding-Orakel-Angriff (Abschnitt 4.5).

5.1.4 Replay-Angriffe

Bei einem Replay-Angriff sendet ein Angreifer zuvor aufgenommene Nachrichten erneut an den Empfänger. Da es sich um eine echte Nachricht handelt, müssen Gegenmaßnahmen getroffen werden, um dieses erneute Senden zu erkennen. Möglichkeiten hierfür sind beispielsweise die Einführung von Sequenznummern wie bei TLS, die Nutzung von Zeitstempeln oder ein Challenge-Response-Verfahren, dass einem Sender eine zufällige Aufgabe stellt, deren Lösung er in der Nachricht mitsenden muss [FSK10].

Die Abwehr von Replay-Angriffen muss in allen Sicherheitsprotokollen bedacht werden.

5.1.5 Kryptographische Prinzipien

TLS bietet auch Beispiele für Prinzipien, die bei der Verwendung von kryptographischen Verfahren zu beachten sind. Auf zwei dieser Beispiele soll genauer eingegangen werden.

Ein Prinzip ist die Reihenfolge, in der Verschlüsselung und Authentifizierung einer Nachricht stattfinden. In TLS wird bei Nutzung von Block- und Stream-Cipher-Suites für eine Nachricht zuerst ein MAC berechnet, der anschließend zusammen mit der Nachricht verschlüsselt wird. Diese Reihenfolge der Operationen wird MAC-then-Encrypt genannt. Laut [BN00] ist Encrypt-then-MAC vorzuziehen. In [Kra01] wird dieses Ergebnis bestätigt, aber auch die Sicherheit von MAC-then-Encrypt unter bestimmten Voraussetzungen gezeigt.
In [FSK10] fügen die Autoren hinzu, dass das Erkennen und Verwerfen ungültiger Nachrichten durch Encrypt-then-MAC vereinfacht wird. Auf der anderen Seite werden aber auch Vorteile der MAC-then-Encrypt-Reihenfolge erwähnt: Die Ein- und Ausgabe der MAC-Funktion sind einem Angreifer verborgen, so dass Angriffe gegen den MAC erschwert werden und die Integrität der Nachricht damit besser geschützt ist. Außerdem wird das Horton-Prinzip angeführt: „Authenticate what is meant, not what is said“ [WS96].
In den meisten Veröffentlichungen wird jedoch die Nutzung von Encrypt-then-MAC gefordert [4].
In TLS hätte die Verwendung dieser Reihenfolge beispielsweise den Padding-Orakel-Angriff (siehe Abschnitt 4.5) verhindert. Auch die Nutzung von AEAD-Chiffren verhindert Probleme mit dieser Reihenfolge, da die Authenzität einer Nachricht bereits durch die Chiffre gewährleistet wird.

Ein weiteres Prinzip ist die Nutzung unvorhersagbarer Initialisierungsvektoren für Blockchiffren im CBC-Modus, die beispielsweise in [FSK10] von den Autoren gefordert wird. In TLS 1.0

4. Siehe beispielsweise auch Moxie Malinspike: The Cryptographic Doom Principle. http://www.thoughtcrime.org/blog/ the-cryptographic-doom-principle/ Abgerufen am 15.10.2015.

wurde der letzte Chiffretextblock als IV genutzt, was den in Abschnitt 4.7 beschriebenen Angriff ermöglichte.

Diese Prinzipien sind nicht TLS-spezifisch, sondern sollten immer beachtet werden, wenn Kryptographie eingesetzt wird.

5.2 Lernen durch Exploration

Wie im letzten Abschnitt deutlich wurde, eignet sich TLS also aus vielerlei Gründen für den Einsatz in der Hochschullehre. Eine spezielle Methode, die in der Lehre zum Einsatz kommen kann und mit der sich diese Arbeit beschäftigen wird, ist die Exploration.

Exploratives Lernen, also „entdeckendes, forschendes oder autonomes Lernen“, empfiehlt sich für Bereiche, deren „Abstraktion oder Komplexität mit anderen Lernmaterialien schwerer erfassbar oder schwerer darstellbar [ist]“ [SS11]. Es wird allerdings geeignete Software (Explorationsmodule bzw. lernförderliche Software) benötigt, die den Lernenden beim explorativen Lernen unterstützt. Hierfür bieten sich oftmals Simulationen an.

> „Simulationen sind Computerprogramme, die Phänomene oder Aktivitäten modellieren und die dafür vorgesehen sind, dass die Nutzer durch Interaktionen mit ihnen etwas über diese Phänomene und Aktivitäten lernen.“ [NDH+08]

Sie ermöglichen es mit dem Lernstoff zu interagieren und Prozesse, die normalerweise nicht beobachtbar sind, sichtbar und erfahrbar zu machen. Durch die Manipulation von dynamischen Elementen können Lernende die Auswirkungen auf das Verhalten eines Systems direkt beobachten [NDH+08].

Erforderlich für erfolgreich zu verwendende Simulationen ist es, Lernenden die Sicht auf sowohl sichtbare als auch nicht sichtbare Konzepte und Komponenten, die in einem System verwendet werden, zu ermöglichen. Dazu ist es immer erforderlich verborgen ablaufende Prozesse an die Benutzeroberfläche der Simulation zu bringen.
Oftmals ist es hilfreich dem Lernenden mehrere Perspektiven auf den Lerngegenstand zu bieten. Ein Beispiel hierfür ist die Möglichkeit bei dem Lernen von objektorientierter Programmierung zwischen einer Quelltextsicht und einem Klassen- oder Objektdiagramm umschalten zu können. Für diese Perspektivenwechsel bieten sich interaktive Lehrmittel an [SS11].

Im weiteren Verlauf dieser Arbeit soll nun eine Anwendung entwickelt werden, die eine per TLS gesicherte Kommunikation simuliert und es so ermöglicht TLS durch exploratives Lernen besser zu verstehen.

6 Anwendung zur TLS-Simulation

Im folgenden Kapitel wird nun auf die im Rahmen dieser Arbeit entwickelte Anwendung eingegangen. Die Anwendung ist speziell für den Einsatz in der Hochschullehre gedacht. Sie soll dazu dienen Protokolle und ihre internen Abläufe darzustellen und durch zusätzliche Interaktivität exploratives Lernen ermöglichen. Eine Vorgabe war die Möglichkeit der Erweiterbarkeit der Anwendung um weitere Protokolle.

Im ersten Abschnitt werden allgemeine Entscheidungen erläutert, die getroffen wurden, um diese Vorgabe zu ermöglichen. Außerdem werden Überlegungen für die Erweiterung der Anwendung um das TLS-Protokoll dargelegt. Danach wird die Implementation beschrieben, wobei zuerst auf die Umsetzung der allgemeinen Anwendung eingegangen wird und danach auf die konkrete Umsetzung des TLS-Protokolls.

In Anhang B werden anhand eines Beispiels einige Hinweise gegeben, was bei der Erweiterung der Anwendung um weitere Protokolle zu beachten ist.

6.1 Analyse und Entwurf

6.1.1 Abstrakter Rahmen für die Anwendung

Um die Vorgabe der Erweiterbarkeit der Anwendung zu ermöglichen, wurden zu Beginn der Entwicklungsarbeit die Gemeinsamkeiten von Protokollen herausgearbeitet, um diese abstrakt umsetzen zu können. So müssen anschließend lediglich protokollspezifische Eigenschaften in Erweiterungen implementiert werden.
Die Anwendung beschränkt sich auf Protokolle mit zwei Parteien. Zwischen diesen Parteien werden Nachrichten eines festen Formats (sogenannte Protokolldateneinheiten) über einen Kanal ausgetauscht. Die Abfolge verschiedener Nachrichtentypen und die bei ihrem Empfang stattfindenden Verarbeitungsschritte sind durch das Protokoll festgelegt. Zusätzlich besitzen die Parteien einen internen Zustand[1], der beispielsweise als Resultat empfangener Nachrichten verändert werden kann.

Aus diesen Voraussetzungen wurde ein abstrakter Rahmen entwickelt, der für konkrete Protokolle erweitert werden muss. Für ein Protokoll sollte eine Art von Nachrichten bestehen, die von den Parteien gesendet und empfangen werden kann. Dazu muss ein Kanal zwischen den Parteien bestehen, über den das Senden geschehen kann.
Um die Parteien, ihre internen Zustände und die Verarbeitungsschritte bei Nachrichtenempfang strukturiert entwickeln zu können, werden die Parteien als endliche Automaten betrachtet. Es können verschiedene Zustände implementiert werden, um strukturiert unterschiedliches Verhalten bei dem Erhalt von Nachrichten zu erlangen. Der Automat besitzt jeweils immer genau einen aktuellen Zustand, an den eingehende Nachrichten weitergeleitet werden. Abhängig

1. Um Verwechslungen mit den im nächsten Absatz eingeführten Automatenzuständen zu vermeiden, wird der Zustand einer Partei im Folgenden immer als interner Zustand bezeichnet.

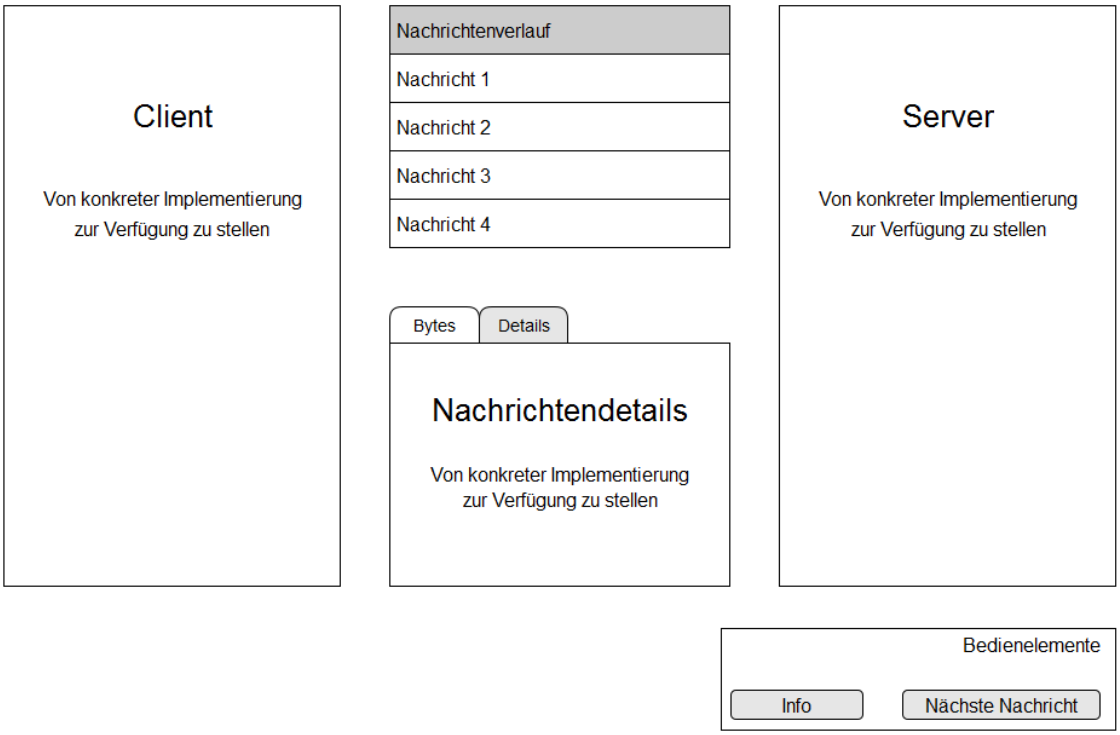

Abbildung 6.1: Prototyp für die Benutzeroberfläche

vom gewünschten Verhalten können aus dem aktuellen Zustand heraus der aktuelle Zustand neu gesetzt und Nachrichten gesendet werden. Der Automat kann seinen Zuständen zusätzlich Zugriff auf den internen Zustand der Partei bereitstellen.

Anschließend wurde prototypisch eine Benutzeroberfläche entwickelt, die auf Basis dieser Überlegungen auch einen visuellen Rahmen bereitstellt, der für konkrete Protokolle gefüllt werden kann. Ein Prototyp ist in Abbildung6.1 zu sehen. Es musste möglich sein, den internen Zustand der beiden Parteien darzustellen, der jedoch vom betrachteten Protokoll abhängig ist. Ebenso sollte zu jeder protokollabhängigen Nachricht, neben ihrer reinen Repräsentation als Bytefolge, auch die Möglichkeit angeboten werden, ihre Bedeutung darzustellen. Zusätzlich sollte es möglich sein, dem Verlauf des Protokoll schrittweise zu folgen, um ablaufende Prozesse und Veränderungen an den internen Zuständen zu erkennen. Für das Verständnis von Protokollverhalten bei (absichtlich oder unabsichtlich) veränderten Nachrichten wäre auch eine Möglichkeit für das Bearbeiten von Nachrichten sinnvoll.
Mit diesen Möglichkeiten werden auch die Forderungen an exploratives Lernen und Simulationen aus Abschnitt 5.2, insbesondere der Perspektivenwechsel, die Anzeige nicht sichtbarer Prozesse und die Interaktivität, erfüllt.

6.1.2 Erweiterung der Anwendung um TLS 1.2

Für diese Arbeit wurde die TLS 1.2-Spezifikation als Erweiterung des abstrakten Rahmens umgesetzt. Hierdurch lässt sich die zum jetzigen Zeitpunkt aktuelle Version explorieren und insbesondere Abwehrmechanismen gegen Angriffe auf ältere Protokollversionen erkennen. Um diese Angriffe beobachten zu können, wäre auch die Umsetzung einer älteren Protokollversion

wie SSL 2.0 nützlich gewesen. Durch die Erweiterbarkeit der Anwendung bietet sich hier die Möglichkeit für spätere Arbeiten.

Im ersten Schritt wurden für Client und Server auf Basis der TLS-Spezifikation Automatenmodelle entwickelt, die abhängig von empfangenen Nachrichten Zustandswechsel und gesendete Nachrichten abbilden. Die Modelle sind im Anhang A zu finden. An den Zustandsübergängen sind jeweils die empfangenen Nachrichten in orange dargestellt, die zu diesem Übergang führen. Ist keine Nachricht angegeben, so findet direkt nach Eintritt in den Zustand der Zustandswechsel statt. Mit OnEnter gekennzeichnete Kästen an Zuständen zeigen in blau Nachrichten, die gesendet werden sollen, wenn ein Zustand aktueller Zustand wird. Kästen an Zustandsübergängen stehen für Aktionen, die von außen von dem Automaten verlangt werden (beispielsweise ein Verbindungsaufbauwunsch). In diese Kästen werden zu sendende Nachrichten ebenfalls in blau dargestellt. In eckigen Klammern werden Bedingungen für Zustandsübergänge und andere Aktionen angegeben. Der Startzustand wird durch die Unterstreichung des Namens gekennzeichnet.

Danach wurde eine einfache Möglichkeit für die Darstellung der internen Zustände von Server und Client sowie für die Bedeutungsdarstellung von Nachrichten ausgearbeitet. Hierbei sollten insbesondere auch die Zustandsveränderungen kenntlich gemacht werden, um abgelaufene Vorgänge zu verdeutlichen. Die Wahl fiel auf eine baumartige Struktur aus Objekten, denen ein Titel und ein Wert sowie eine optionale Liste von Kindelementen zugewiesen werden kann. Veränderungen zwischen zwei Zuständen sollten durch Kennzeichnung der geänderten Objekte geschehen.

Anschließend wurde die TLS-Spezifikation auf Funktionen untersucht, die nicht implementiert werden sollten. Dies geschah aus Gründen der Komplexitätsreduktion zum besseren Verständnis, insbesondere wenn die Funktion wenig dazu beigetragen hätte. Im Folgenden soll kurz auf diese Funktionen und auf die Gründe, aus denen sie ausgelassen wurden, eingegangen werden.

Auf TLS-Extensions wurde komplett verzichtet, da sie nicht für das grundsätzliche Verständnis von TLS notwendig sind, sondern eher technische Erweiterungen des Standards darstellen, um bestimmte Funktionen nachzurüsten oder Verhalten zu erzwingen.

Auch die Möglichkeit TLS-Plaintexte vor dem Verschlüsseln zu komprimieren wurde nicht implementiert, da die Funktion wenig zum Verständnis beiträgt und zusätzlich aufgrund ihrer Angreifbarkeit in der nächsten Protokollversion wahrscheinlich nicht mehr enthalten sein wird (vgl. Abschnitt 4.9).

TLS unterstützt optional zusätzliche Clientauthentifizierung. Diese wird allerdings in vielen Anwendungsfällen, wie bei dem verschlüsselten Abruf einer Internetseite über HTTPS, nicht genutzt und unterscheidet sich nicht übermäßig von der Serverauthentifizierung. Daher wurde auch auf diese Funktion in der Anwendung verzichtet.

Weiterhin wurden nur einige Cipher-Suites implementiert. Hierbei wurde jedoch darauf geachtet, dass sowohl Block- und AEAD-Chiffren zur Ver- und Entschlüsselung als auch RSA und das Diffie-Hellman-Verfahren zum Schlüsselaustausch verwendet wurden. Auf die Implementierung von Cipher-Suites, die sich lediglich in Schlüssellängen von bereits implementierten Cipher-Suites unterscheiden oder deren Chiffren bzw. Schlüsselaustauschalgorithmen bereits durch andere Cipher-Suites abgedeckt waren, wurde bewusst verzichtet. Aus Zeitgründen und da von der Verwendung bereits abgeraten wird, wurde auch auf die Implementierung einer Cipher-Suite mit der Stromchiffre RC4 verzichtet.

Auch die sichere Verwaltung von Schlüsseln im Dateisystem, sowie die Entfernung verwendeter Schlüssel aus dem Speicher, auf die in produktiv verwendeten Systemen geachtet werden muss, wurde hier aus naheliegenden Gründen vernachlässigt.

Zusätzlich wurde auch auf die Möglichkeit der Übertragung eigener Daten oder Simulation von auf TLS aufsetzenden Protokollen wie HTTPS bei bestehender Verbindung verzichtet. Diese Funktion wäre jedoch beispielsweise für Betrachtungen zur Erkennung der Nachrichtenlänge oder zur Darstellung von Angriffen, wie den in den Abschnitten 4.8 und 4.9 beschriebenen, hilfreich.

Aus Zeitgründen wurde auch auf die Wiederaufnahme bestehender Sitzungen verzichtet (siehe Abschnitt 3.5). Der verkürzte Handshake hat eher praktische Bedeutung, da auf erneute Aushandlung kryptographischer Verfahren und damit mehrere Nachrichten während des Handshakes verzichtet werden kann und die Wiederaufnahme der Verbindung somit weniger Zeit in Anspruch nimmt. Eine spätere Erweiterung der Anwendung um diese Funktion wäre jedoch sinnvoll, um das Verständnis von TLS zu vertiefen.

Ein letzter Punkt, der in der Anwendung nicht implementiert wurde, ist die Zertifikatsvalidierung. Diese ist zwar entscheidend für den sicheren Aufbau einer Verbindung, hat aber eher indirekt mit der der Protokollspezifikation zu tun. Daher wurde aus Zeitgründen auf diese Funktion verzichtet, die sich jedoch für spätere Arbeiten anbietet.

6.2 Implementierung

Die Wahl der Programmiersprache für die Anwendung fiel auf Java, da diese Sprache an der Universität Hamburg in der ersten Semestern zum Einsatz kommt und von den Studierenden zwingend gelernt werden muss. Auf diese Weise sollte die entwickelte Anwendung auch von anderen Studierenden erweitert oder zumindest problemlos verstanden werden können. Zur Erzeugung der Benutzeroberfläche wurde die in der Java-Runtime verfügbare Bibliothek Swing genutzt, da diese ebenfalls in einigen Veranstaltungen an der Universität Hamburg verwendet wird.

6.2.1 Abstrakter Rahmen für die Anwendung

Ein UML-Diagramm der entwickelten Klassen für den abstrakten Rahmen ist in Abbildung 6.2 zu finden. Im Folgenden werden diese Klassen im Detail betrachtet.

Protokollnachrichten werden durch die abstrakte Klasse `ProtocolDataUnit` dargestellt. In Unterklassen muss die Byte-Repräsentation einer Nachricht sowie Titel und Untertitel für die Darstellung auf der Benutzeroberfläche implementiert werden.

Für den abstrakten Rahmen wurden die Parteien als endliche Automaten modelliert. Für diese Modellierung wurde das Entwurfsmuster Zustand eingesetzt. Details hierzu sind [FF04] zu entnehmen.
Die abstrakte Klasse `StateMachine` bildet das Grundgerüst für einen Automaten und die abstrakte Klasse `State` wurde zur Implementierung der jeweiligen Automatenzustände genutzt. In einem `StateMachine`-Objekt kann der aktuelle Zustand gesetzt und abgefragt, sowie Nachrichten gesendet werden. Die möglichen Zustände müssen dem Objekt vorher mitgeteilt

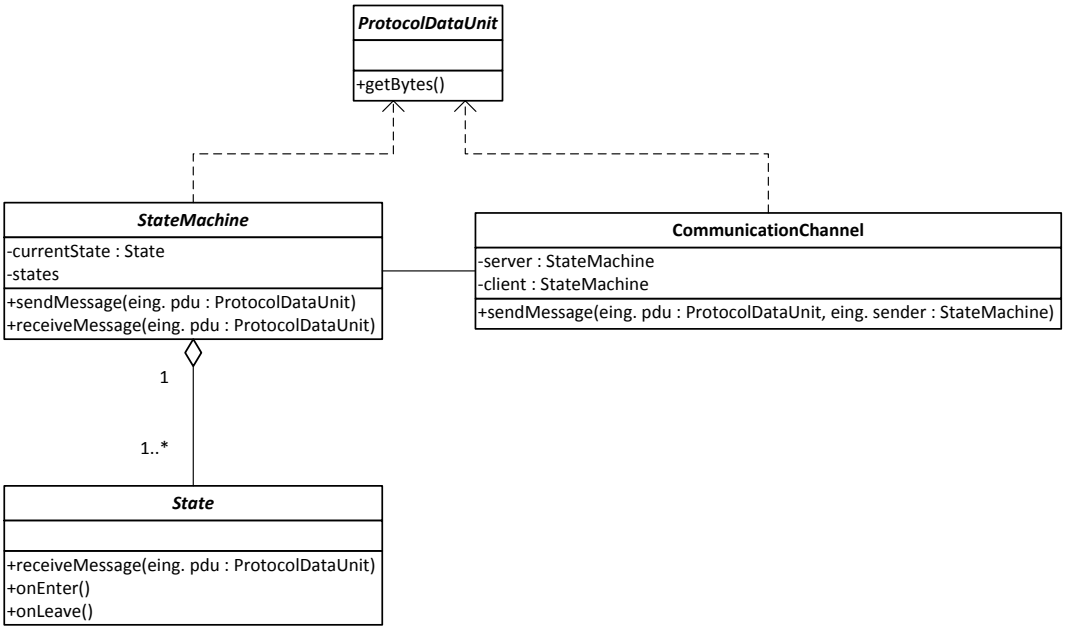

Abbildung 6.2: UML-Diagramm der Umsetzung des abstrakten Rahmens

werden. Zur Beobachtung von Zustandsänderungen wurde das Beobachter-Muster verwendet. Dieses Entwurfsmuster ist ebenfalls in [FF04] zu finden.
Unterklassen von `State` implementieren das Verhalten bei Empfang einer Nachricht und bei Betreten und Verlassen des Zustand. Außerdem ermöglichen sie das Senden von Nachrichten und das Setzen des aktuellen Zustands des besitzenden `StateMachine`-Objekts.

Für die Übertragung von Nachrichten wird ein `CommunicationChannel`-Objekt genutzt, dass die Übermittlung von Nachrichten zwischen den beiden an dem Protokollablauf beteiligten `StateMachine`-Objekten übernimmt. Außerdem ist in diesem Objekt auch die Thread-Synchronisierung implementiert, die dafür sorgt, dass der Protokollablauf beim Senden einer Nachricht pausiert wird. Dies ermöglicht das schrittweise Verfolgen der gesendeten Nachrichten und der Veränderungen des internen Zustands der Kommunikationspartner. Auch `CommunicationChannel`-Objekte können beobachtet werden, um über zu sendende oder gesendete Nachrichten informiert zu werden.

Um Protokollerweiterungen eine typsichere Umgebung zu bieten, besitzen die `StateMachine`-, `State`- und `CommunicationChannel`-Klassen einen generischen Parameter vom Typ einer Unterklasse von `ProtocolDataUnit`. Hierdurch können Unterklassen ihren Nachrichtentyp angeben und sich auf die durch Java gewährleistete Typsicherheit verlassen. listing of headers?

Abstrakte Benutzeroberfläche

Für die Darstellung des Protokollablaufs wurde ein Gerüst entwickelt, das die im vorherigen Abschnitt beschriebenen Gemeinsamkeiten von Protokollen abbildet. Für die internen Zustände der Parteien wurden zwei Bereiche angelegt, die durch konkrete Protokolle bereitgestellt werden müssen. Außerdem wird eine Liste von bereits gesendeten bzw. als nächstes gesendeten Nachrichten angezeigt, die es bei Auswahl einer Nachricht ermöglicht, ihre Details anzuzeigen. Dabei wird für jede Nachricht die Darstellung als Bytekette und eine protokollspezifische Detailansicht, die ebenfalls durch konkrete Protokolle bereitgestellt werden muss, angeboten. Zusätzlich

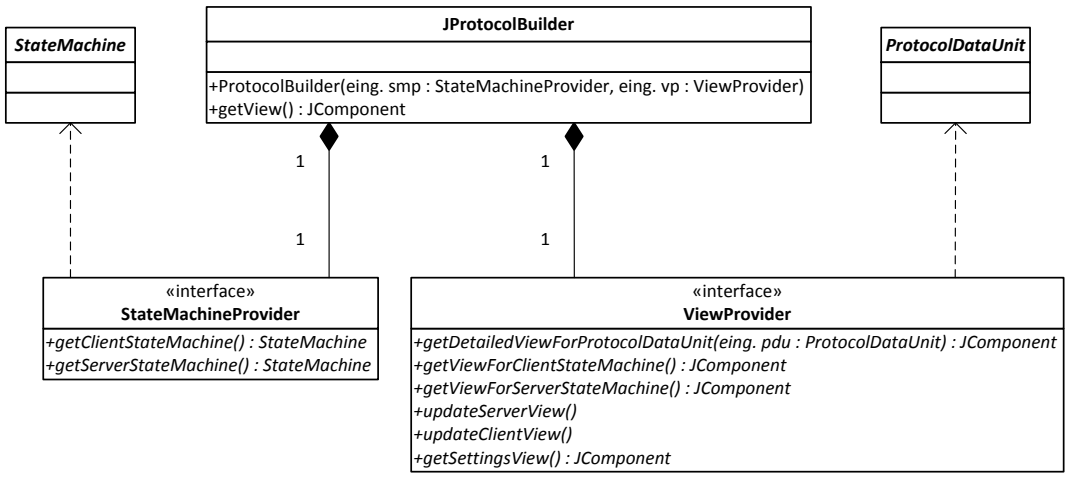

Abbildung 6.3: UML-Diagramm von Provider- und ProtocolBuilder-Klassen für die Bereitstellung von Erweiterungen

kann ein weiteres Fenster angezeigt werden, dass es ermöglicht Informationen zu gewünschten zustands- oder nachrichtenabhängigen Daten anzubieten. Für die Interaktion wurde ein Bedienfeld entwickelt, dass die Nachrichtensteuerung bereitstellt und das Informationsfenster öffnet. In der Nachrichtendetaildarstellung können die Bytes einer Nachricht vor dem Senden verändert werden.

Für die Verknüpfung der Datenschicht mit der Benutzeroberfläche wurde das Entwurfsmuster Model-View-Presenter (siehe [Pot96]) gewählt, um eine strukturierte Trennung der Schichten zu ermöglichen.

Erweiterung der Anwendung

Um Erweiterungen die Bereitstellung fehlender Elemente und nötiger Objekte zu ermöglichen, wurden die Interfaces `ViewProvider` und `StateMachineProvider` erstellt, die eine Erweiterung implementieren muss. Objekte, die diese Interfaces implementieren, ermöglichen es, das Gerüst mit protokollspezifischen Details zu füllen. `ViewProvider` sorgen für die Bereitstellung von Fensterelementen für die internen Zustände der Parteien, für die Detailansicht einer Nachricht und für optionale Einstellmöglichkeiten zu dem Protokollablauf. `StateMachineProvider` sind zuständig für die entsprechenden `StateMachine`-Objekte für die beiden Parteien. Diese werden dann durch einen `ProtocolBuilder` mit einem `CommunicationChannel` und den entsprechenden Fensterelementen verknüpft. In Abbildung 6.3 ist ein UML-Diagramm zu finden, das diese Zusammenhänge visualisiert.

Beim Laden einer neuen Protokollsimulation wird eine Liste vorhandener Protokolle und bei Auswahl eines Eintrags die entsprechende Einstellungsansicht angezeigt. Wenn eine Protokollsimulation gestartet wird, wird das vom `ProtocolBuilder` zur Verfügung gestellte Fensterelement in die Anwendung eingebettet.

6.2.2 Erweiterung der Anwendung um TLS 1.2

Nachrichten

Als Protokollnachricht dient die Klasse `TlsCiphertext`. Bei ihrer Modellierung wurde versucht analog zu der Spezifikation vorzugehen und auch Benennungen möglichst einheitlich zu halten. Sie enthält Felder für den `TlsContentType`, die `TlsVersion` und ihre Länge, sowie ein `TlsFragment`, das die eigentliche Nachricht der oberen TLS-Protokollschicht enthält. Abhängig von der aktuellen Cipher-Suite wird dieses `TlsFragment` verschlüsselt und mit einem MAC versehen. Zusätzlich kann es weitere Felder für bei der Verschlüsselung benötigte Werte enthalten (siehe Abschnitt 3.2 und insbesondere Abbildung 3.2).

Die Nachrichten der oberen Schicht werden durch für jedes Teilprotokoll bestehende Unterklassen von `TlsMessage` dargestellt. Neben `TlsAlertMessage`, `TlsChangeCipherSpecMessage` und `TlsApplicationDataMessage` wurde zusätzlich für jeden HandshakeType eine eigene Klasse erstellt. Alle diese Nachrichten enthalten zwei Konstruktoren, die die Erzeugung sowohl aus benötigten Werten als auch durch das Parsen übertragener Bytes ermöglichen.

Automatenimplementierung

Im Folgenden werden die implementierten Klassen für das TLS-Automatenmodell beschrieben. Ein UML-Diagramm ist in Abbildung 6.4 zu finden.

Als Unterklasse der `StateMachine`-Klasse wurde `TlsStateMachine` implementiert. Diese Klasse bietet Zugriff auf alle notwendigen, internen Zustandsinformationen von TLS-Client und -Server. Ein Objekt vom Typ `TlsSecurityParameters` bietet Zugriff auf während des Handshakes ausgehandelte Verfahren und Informationen wie SessionId, Random-Werte und das berechnete MasterSecret. Objekte vom Type `TlsConnectionState` bilden den current read und write state, sowie den pending state (vgl. Abschnitt 3.6). In ihnen werden ausgehandelte Cipher-Suite und Schlüssel sowie die aktuelle Sequenznummer verwaltet. Bei Empfang oder nach dem Senden einer ChangeCipherSpec-Nachricht kann der pending state zum current state gemacht werden.
Für Client und Server wurde jeweils eine Unterklasse `TlsClientStateMachine` und `TlsServerStateMachine` erstellt, die unterschiedliches Verhalten der beiden Parteien spezifizieren. Insbesondere im Bereich der asymmetrischen Kryptographie sind diese Unterschiede so groß, dass der Mehraufwand gerechtfertigt ist. Beispielsweise besitzt der Server ein RSA-Schlüsselpaar, dessen geheimer Schlüssel zum Entschlüsseln des vom Client ausgewählten PreMasterSecret oder zum Signieren der Diffie-Hellman-Parameter genutzt wird. Der Client ist im Gegensatz dazu lediglich im Besitz des öffentlichen Serverschlüssels zum Verschlüsseln des PreMasterSecrets oder zum Überprüfen der Parametersignatur.

Entsprechend den Automatenmodellen im Anhang A wurden für alle Zustände entsprechende Unterklassen von `TlsState`, einer Unterklasse von `State`, erstellt. Jede dieser Unterklassen definiert die Typen von TLS-Nachrichten der oberen Schicht, die sie erwartet, und implementiert das Verhalten bei Empfang einer solchen Nachricht. Beispielsweise erwartet der Clientzustand `TlsWaitingForServerHelloState` lediglich Handshake-Nachrichten vom Typ ServerHello. Bei Empfang einer solchen Nachricht werden die entsprechenden Felder für TlsVersion, ServerRandom, SessionID und Cipher-Suite gesetzt und der aktuelle Zustand der `TlsClientStateMachine` auf den nachfolgenden Zustand gesetzt. Diese Umsetzung ist

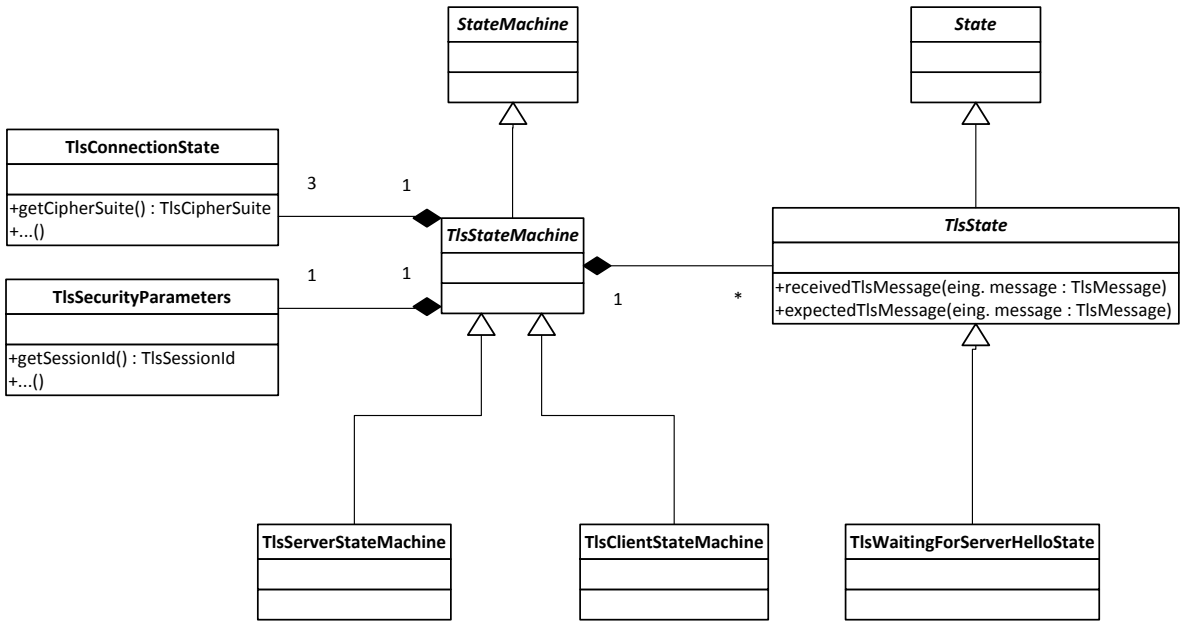

Abbildung 6.4: UML-Diagramm der Klassen für das Tls-Automatenmodell

in Listing 6.1 zu finden. Nach dem gleichen Prinzip wurde das Verhalten in allen anderen Zuständen implementiert.

```
1  public class TlsWaitingForServerHelloState extends TlsState {
2      ...
3      @Override
4      public boolean expectedTlsMessage(TlsMessage message) {
5          return isHandshakeMessageOfType(message,
               TlsHandshakeType.server_hello);
6      }
7
8      @Override
9      public void receivedTlsMessage(TlsMessage message) {
10         ...
11         setServerHelloValues(message);
12         setTlsState(TlsStateType.CLIENT_IS_WAITING_FOR_CERTIFICATE_STATE)
13     }
14
15     private void setServerHelloValues(TlsServerHelloMessage message) {
16         ...
17         _stateMachine.setVersion(version);
18         _stateMachine.setServerRandom(message.getServerRandom());
19         _stateMachine.setSessionId(message.getSessionId());
20         _stateMachine.setPendingCipherSuite(message.getCipherSuite());
21     }
22 }
```

Listing 6.1: Implementierung des Zustands `TlsWaitingForServerHelloState`

In der Oberklasse `TlsState` werden eingehende Nachrichten entschlüsselt. Dieser Vorgang wird in den folgenden Absätzen näher beleuchtet. Danach werden die Nachrichten je nach gesetztem ContentType geparst. Fehler bei diesem Vorgang führen zu einem Übergang in einen Fehlerzustand und Senden einer Alert-Nachricht. Anschließend wird überprüft, ob die geparste

Nachricht von dem aktuellen Zustand erwartet wird. Ist dies nicht der Fall, so wird ebenfalls ein Fehlerzustand gesetzt und eine Alert-Nachricht gesendet.

Kryptographie

Im TLS-Protokoll werden die verwendeten Algorithmen und Schlüssellängen durch Cipher-Suites zur Verfügung gestellt (siehe Abschnitt 3.9). Durch das Interface `TlsCipherSuite` werden diese Werte zur Verfügung gestellt. Für die Anwendung wurden verschiedene `TlsCiphersuite`-implementierende Klassen entwickelt, die verschiedene Algorithmen zum Schlüsselaustausch und zur Verschlüsselung beinhalten:

- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_128_GCM_SHA256

Durch diese Wahl der Cipher-Suites können sowohl der Diffie-Hellman- als auch der RSA-Schlüsselaustausch beobachtet werden. Außerdem ist die Verschlüsselung per Blockchiffre (AES im CBC-Modus) oder AEAD-Chiffre (AES im GCM-Modus) möglich.

Für die kryptographischen Algorithmen wurde auf die Implementierungen der Java-Bibliothek zurückgegriffen. Die TLS-spezifische PseudoRandomFunction wurde eigenständig entwickelt und durch Testvektoren auf ihre Richtigkeit hin überprüft. Ebenso wurde auch die Verknüpfung der Algorithmen für die Ver- und Entschlüsselung, also beispielsweise die Berechnung von MAC, Padding und Paddinglänge und die anschließende Verschlüsselung, für Block- und AEAD-Chiffren selber in den Klassen `TlsBlockCipherSuite` und `TlsAeadCipherSuite` implementiert. Konkrete Cipher-Suites füllen lediglich die vorgegebenen Rahmen und bieten Zugriff auf Schlüssellängen und die verwendeten Algorithmen.

In der `TlsState`-Klasse werden zu sendende Nachrichten durch die im current write state gesetzte Cipher-Suite von `TlsPlaintext`- in `TlsCiphertext`-Objekte umgewandelt und anschließend gesendet. Beim Empfang von Nachrichten wird dieser Prozess umgekehrt durchlaufen. Bei auftretenden Fehlern wie einem falschen Schlüssel oder einem ungültigen MAC erfolgt wiederum der Übergang in einen Fehlerzustand und das Senden einer Alert-Nachricht.

Benutzeroberfläche

Für die Benutzeroberfläche wurde den Überlegungen im vorherigen Abschnitt entsprechend für Client und Server ein auf der Swing-Komponente `JTree` basierendes Fensterelement entwickelt. Der interne Zustand der Parteien wird durch Titel-Wert-Paare und optionale Kindelemente dargestellt. Die Klasse `TlsStateMachine` enthält eine Methode, die die Paare für den aktuellen Zustand zurückliefert. Bei Zustandsänderungen, die durch das Beobachter-Muster kommuniziert werden, werden der alte und neue Zustand verglichen und neue oder geänderte Einträge gekennzeichnet, sodass Auswirkungen von empfangenen Nachrichten sichtbar werden. Zusätzlich wurde zu den jeweiligen Einträgen jeweils ein erklärender Informationstext mit Auszügen aus der Spezifikation erstellt, um ein besseres Verständnis der Abläufe zu unterstützen.

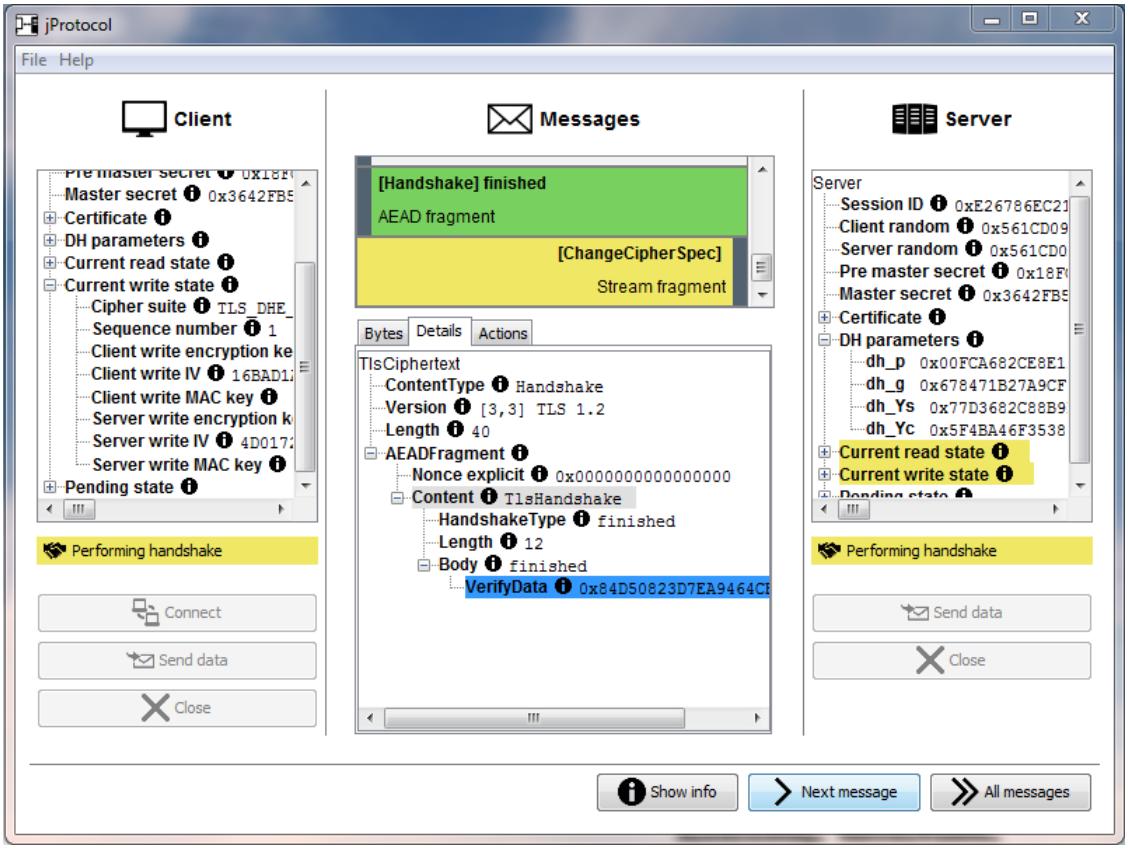

Abbildung 6.5: Die fertige Anwendung

Zusätzlich enthalten die Fensterelemente Buttons für die verfügbaren Aktionen wie das Verbinden oder das Senden von Daten bei bestehender Verbindung.

Das gleiche Fensterelement wie für Server und Client wurde auch für die Detaildarstellung von Nachrichten verwendet. Auch für die jeweiligen Nachrichtenfelder wurden Informationstexte erstellt, die beim Verstehen der Protokolls helfen sollen.

Ein Screenshot der endgültigen Anwendung mit der Erweiterung um TLS ist in Abbildung 6.5 zu finden.

7 Fazit

In dieser Arbeit wurde gezeigt, dass das TLS-Protokoll sich für den Einsatz in der Hochschullehre im Bereich der IT-Sicherheit eignet. Hierfür spricht insbesondere die Möglichkeit viele Grundlagen der IT-Sicherheit und Kryptographie am Beispiel von TLS erklären zu können, die in vielen Anwendungen und Bereichen und voraussichtlich auch in den nächsten Jahren ihre Bedeutung nicht verlieren werden. Diese Grundlagen wurden in dieser Arbeit betrachtet. Die auf Empfehlungen der Fachdidaktik zu explorativem Lernen und Simulationen basierend entwickelte Anwendung unterstützt das Verständnis der Funktionsweise von TLS.

umformulieren

Weiterführende Arbeiten könnten die Anwendung auf zweierlei Arten erweitern. Erstens wäre die Implementierung einiger nicht umgesetzter Funktionen, die in Abschnitt 6.1.2 beschrieben wurden, sinnvoll. Gerade die Implementierung des verkürzten Handshakes, der Validierung von Zertifikaten und das Versenden unterschiedlicher Daten würden sich hier anbieten.
Zweitens bietet der modulare und erweiterbar gehaltene Aufbau der Anwendung die Möglichkeit auch andere Protokolle wie IPSec zu implementieren. Aber auch das Verständnis anderer Verfahren wie beispielsweise Challenge-Response-Authentication könnte durch die Anwendung erleichtert werden. Als Hilfestellung wird in Anhang B dieser Arbeit anhand eines einfachen Beispiels auf Dinge hingewiesen, die bei der Erweiterung der Anwendung beachtet werden müssen.

Literaturverzeichnis

[ABD+] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. https://weakdh.org/imperfect-forward-secrecy.pdf. Zugriff am 22.05.2015.

[AFP13] N. J. Al Fardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. *2014 IEEE Symposium on Security and Privacy*, pages 526–540, 2013.

[Bar04] Gregory V. Bard. The Vulnerability of SSL to Chosen Plaintext Attack, 2004.

[Bar06] Gregory V. Bard. A Challenging But Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL. In *SECRYPT 2006, Proceedings of the international conference on security and cryptography*, pages 7–10. INSTICC Press, 2006.

[BDLF+] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Santiago Zanella-Béguelin, Jean-Karim Zinzindohoué, and Benjamin Beurdouche. FREAK: Factoring RSA Export Keys. https://www.smacktls.com/#freak. Zugriff am 22.05.2015.

[Ble98] Daniel Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '98, pages 1–12, London, UK, 1998. Springer-Verlag.

[BN00] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In *Advances in Cryptology - ASIACRYPT 2000*. Springer Berlin Heidelberg, 2000.

[BTPL15] R. Barnes, M. Thomson, A. Pironti, and A. Langley. Deprecating Secure Sockets Layer Version 3.0. RFC 7568, 2015.

[CHVV03] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password Interception in a SSL/TLS Channel. In *Advances in Cryptology - CRYPTO 2003*, volume 2729, pages 583–599. Springer, 2003.

[DA99] T. Dierks and C. Allen. The TLS Protocol - Version 1.0. RFC 2246, 1999.

[DR06] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol - Version 1.1. RFC 4346, 2006.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol - Version 1.2. RFC 5246, 2008.

[DR11] Thai Duong and Juliano Rizzo. Here Come The XOR Ninjas. 2011.

[Eck13] Claudia Eckert. *IT-Sicherheit - Konzepte, Verfahren, Protokolle*. Oldenbourg Verlag, München, 2013.

[FF04] Eric Freeman and Elizabeth Freeman. *Head First Design Patterns*. O'Reilly Media, Sebastopol (Kalifornien), 2004.

[FKK11] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, 2011.

[FSK10] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.

[JK03] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, 2003.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, 1997.

[KK12] Christina Klüver and Jürgen Klüver. *Lehren, Lernen und Fachdidaktik - Theorie, Praxis und Forschungsergebnisse am Beispiel der Informatik*. Springer Vieweg, Wiesbaden, 2012.

[Kra01] Hugo Krawczyk. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 310–331, London, UK, 2001. Springer-Verlag.

[Kra10] Hugo Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 631–648, 2010.

[MDK14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE Bites: Exploiting the SSL 3.0 Fallback, 2014.

[Mey14] Christopher Meyer. *20 Years of SSL/TLS Research - An Analysis of the Internet's Security Foundation*. Dissertation, Ruhr-University Bochum, 2014.

[MS13] Christopher Meyer and Jörg Schwenk. SoK: Lessons Learned From SSL/TLS Attacks. In *The 14th International Workshop on Information Security Applications*, 2013.

[NDH+08] Helmut M. Niegemann, Steffi Domagk, Silvia Hessel, Alexandra Hein, Matthias Hupfer, and Annett Zobel. *Kompendium multimediales Lernen*. Springer Verlag, Berlin Heidelberg, 2008.

[Pot96] Mike Potel. MVP: Model-View-Presenter - The Taligent Programming Model for C++ and Java . http://www.wildcrest.com/Potel/Portfolio/mvp.pdf, 1996. Zugriff am 13.10.2015.

[Res15] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 - draft-ietf-tls-tls13-07. Technical report, 2015.

[Sch06] Bruce Schneier. *Angewandte Kryptographie - Der Klassiker. Protokolle, Algorithmen und Sourcecode in C*. Pearson Studium, München, 2006.

[Sch09] Klaus Schmeh. *Kryptographie - Verfahren, Protokolle, Infrastrukturen*. dpunkt.verlag, Heidelberg, 2009.

[SS11] Sigrid Schubert and Andreas Schwill. *Didaktik der Informatik*. Spektrum Akademischer Verlag, Heidelberg, 2011.

[TP11] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176, 2011.

[Vau02] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology - EUROCRYPT 2002*, pages 534–545. Springer, 2002.

[WS96] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 Protocol. In *The Second USENIX Workshop on Electronic Commerce*, pages 29–40. USENIX Association, 1996.

A TLS-Automatenmodelle

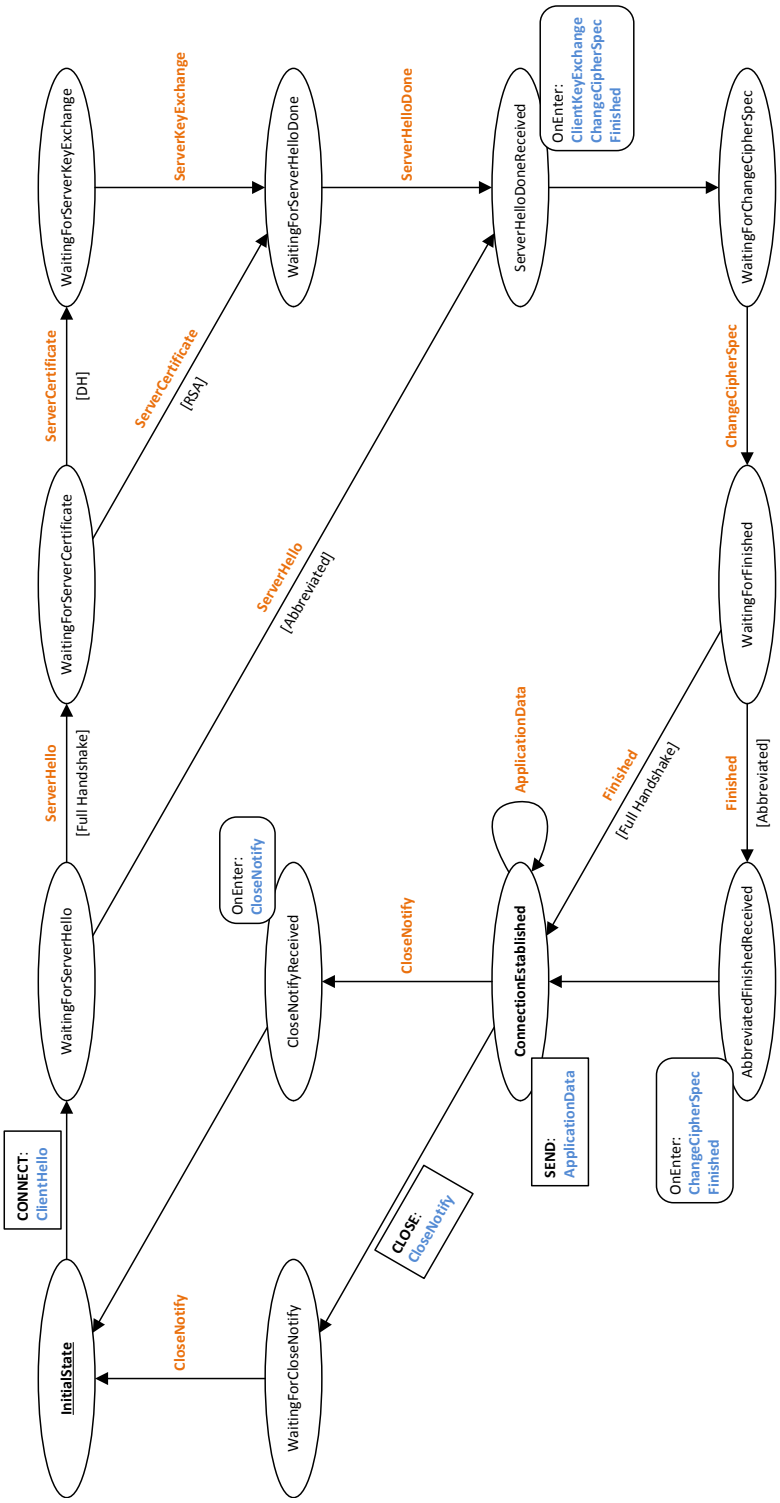

Abbildung A.1: Das Automatenmodell für den TLS-Client

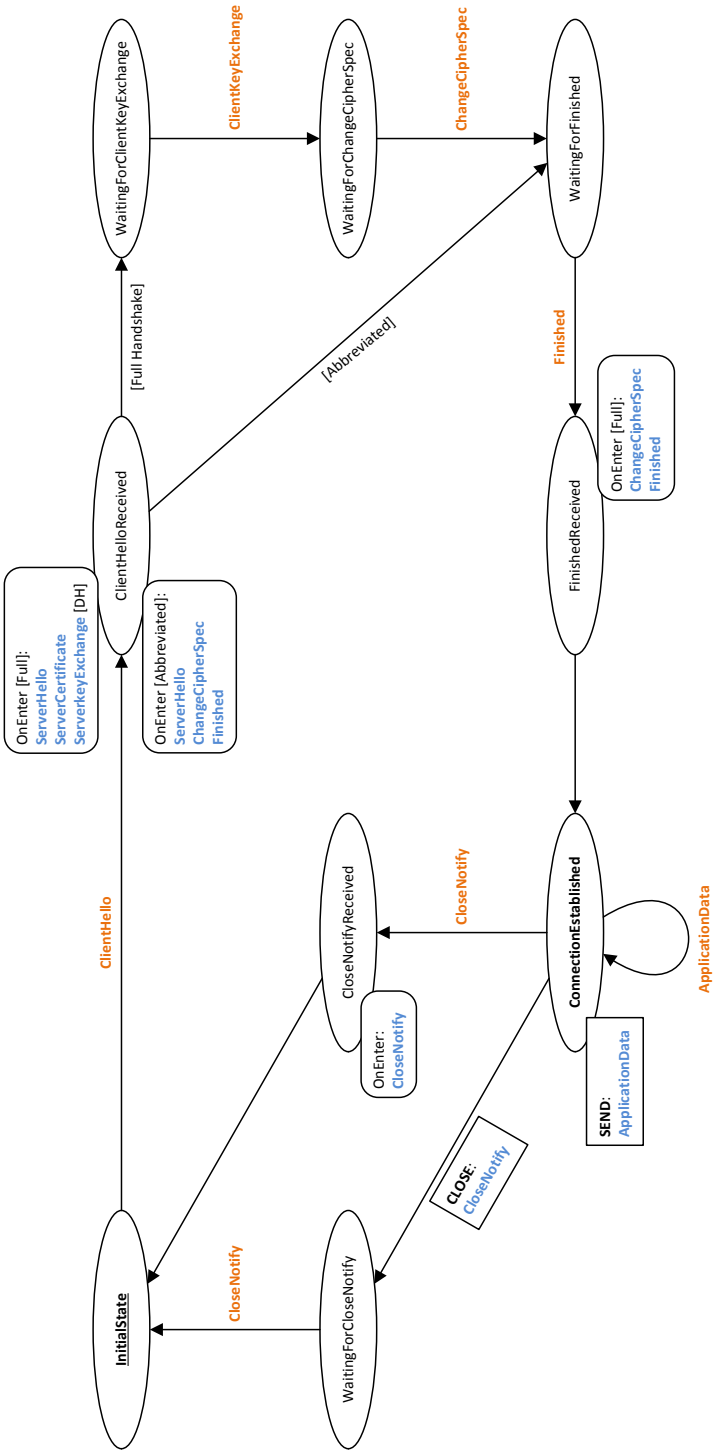

Abbildung A.2: Das Automatenmodell für den TLS-Server

B Erweiterung der Anwendung

In diesem Abschnitt soll anhand eines einfachen Beispiels gezeigt werden, was bei der Erweiterung der Anwendung um ein weiteres Protokoll zu beachten ist. Dazu wird ein einfaches Protokoll betrachtet, bei dem eine Nachricht lediglich aus einem Längenfeld und einem Nachrichteninhalt besteht. Ein Client kann Nachrichten dieses Formats an den Server senden, der diese lediglich genauso zurücksendet. Es handelt sich also um eine leicht modifizierte Variante des Echo-Protokolls[1].

Im ersten Schritt wird der Nachrichtentyp implementiert (siehe Listing B.1).

```
1  public class EchoMessage extends ProtocolDataUnit {
2
3      private String _messageContent;
4      ...
5      @Override
6      public byte[] getMessageBytes() {
7          byte[] lengthBytes = ByteHelper.intToByteArray(getLength());
8          byte[] messageContentBytes =
               _messageContent.getBytes(StandardCharsets.US_ASCII);
9          return ByteHelper.concatenate(lengthBytes,
               messageContentBytes);
10     }
11
12     @Override
13     public String getTitle() {
14         return "Echo message";
15     }
16
17     @Override
18     public String getSubtitle() {
19         return _messageContent;
20     }
21 }
```

Listing B.1: Implementierung des Nachrichtentyps `EchoMessage`

Anschließend werden Automaten für Server und Client entwickelt. Dazu wird die Klasse `StateMachine` mit dem formalen Typparameter `EchoMessage` erweitert. Für die Automaten müssen von `State` abgeleitete Zustände entwickelt werden, die das Verhalten abbilden. In diesem einfachen Beispiel genügt ein Zustand für den Server, in dem empfangene Nachrichten unverändert wieder gesendet werden (siehe Listing B.2). Analog ist für den Client vorzugehen. Der Client benötigt jedoch zusätzlich noch eine Methode `sendEchoRequest()`, um Nachrichten an der Server senden zu können. Außerdem wurden, um den Zustandswechsel zu demonstrieren, ein Sende- und ein Empfangszustand implementiert.

```
1  public class EchoServer extends StateMachine<EchoMessage> {
2
3      public final Integer RECEIVE_STATE = 1;
```

1. Das Echo-Protokoll wird in RFC 862 spezifiziert.

```
4
5       public EchoServer() {
6           addState(RECEIVE_STATE, new ReceiveState(this));
7           setState(RECEIVE_STATE);
8       }
9   }
10
11  public class ReceiveState extends State<EchoMessage> {
12
13      public ReceiveState(EchoServer stateMachine) {
14          super(stateMachine);
15      }
16
17      @Override
18      public void receiveMessage(EchoMessage pdu) {
19              EchoMessage echo = new
                    EchoMessage(pdu.getMessageContent());
20              sendMessage(echo);
21      }
22  }
```

Listing B.2: Implementierung des Serverautomaten

Danach wird eine Klasse `EchoProvider` erstellt, die der Anwendung Zugriff auf die Automaten für Server und Client, sowie auf zu implementierende Fensterelemente bietet. Außerdem muss sich die Klasse auch um die Aktualisierung der Fensterelemente bei Änderung des internen Zustand von Server oder Client kümmern. Diese Klasse muss die Interfaces `ViewProvider` und `StateMachineProvider` implementieren (siehe Listing B.3).

```
1   public class EchoProvider implements ViewProvider<EchoMessage>,
        StateMachineProvider<EchoMessage> {
2
3       @Override
4       public StateMachine<EchoMessage> getServerStateMachine() {
5           return new EchoServer();
6       }
7       ...
8   }
```

Listing B.3: Implementierung der `EchoProvider`-Klasse

Für den Server wird lediglich ein leeres Fensterelement erzeugt, für den Client eine Liste mit empfangenen Serverantworten sowie ein Button, um neue Anfragen abzuschicken. Zwei Dinge sind hier zu beachten.
Methoden, die aus dem UI-Thread (also beispielsweise bei Drücken eines Buttons) aufgerufen werden und dazu führen, dass eine neue Nachricht gesendet wird, müssen zwingend in einem neuen Thread aufgerufen werden, um das erfolgreiche Pausieren bei einer Nachrichtenübertragung zu ermöglichen. Die Umsetzung für den Echo-Client ist in Listing B.4 zu finden.

```
1   JButton sendButton = new JButton("Send...");
2   sendButton.addActionListener(new ActionListener() {
3       @Override
4       public void actionPerformed(ActionEvent e) {
5           new Thread(new Runnable() {
6               @Override
7               public void run() {
8                   _client.sendEchoRequest("DATEN!");
```

```
            }
        }).start();
    }
});
```

Listing B.4: Starten eines neuen Threads für das Senden einer Nachricht

Außerdem kann ein bereits implementierter Mechanismus dazu genutzt werden, auf Zustandsänderungen von Server oder Client zu reagieren. Dazu muss lediglich die Methode `notifyObserversOfStat` aus einem Zustand heraus aufgerufen werden. Dies führt zur Ausführung von `updateServerView` bzw. `updateClientView` des `EchoProvider`-Objekts, in dem die erstellten Fensterelemente aktualisiert werden können. In der `updateClientView`-Methode der `EchoProvider`-Klasse wird beispielsweise die Liste der bisher empfangenen Serverantworten aktualisiert (siehe Listing B.5).

```
class ReceiveState extends State<EchoMessage> {
    ...
    @Override
    public void receiveMessage(EchoMessage pdu) {
        _receivedMessages.add(pdu.getPayload());
        _stateMachine.notifyObserversOfStateChanged();

        setState(SEND_STATE);
    }
}

public class EchoProvider implements ... {
    ...
    @Override
    public void updateClientView() {
        _clientTextArea.setText("");
        for (String text : _client.getReceivedMessages()) {
            _clientTextArea.append(text + "\n");
        }
    }
}
```

Listing B.5: Aktualisierungsmechanismus bei Zustandsänderung

Als letztes muss das entwickelte Protokoll noch in der Anwendung einzutragen. Dazu muss der Protokollliste in der Klasse `ProtocolRegistry` noch ein `JProtocolBuilder`-Objekt hinzugefügt werden (siehe Listing B.6).

```
public class ProtocolRegistry {
    ...
    public ProtocolRegistry() {
        ...
        EchoProvider echoProvider = new EchoProvider();
        _protocolMap.put("EchoProtocol", new
            ProtocolBuilder<>(echoProvider, echoProvider));
    }
}
```

Listing B.6: Protokollregistrierung

Ein abschließender Hinweis: Für die Darstellung von Client, Server und den Details einer Nachricht kann auch die in dem entwickelten TLS-Plugin verwendete Baumstruktur verwendet werden, die als Klasse `KeyValueTree` verfügbar ist.

Selbständigkeitserklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit der Einstellung der Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den 16. Oktober 2015

Tom Petersen

Todo list