NTNU
Norwegian University of
Science and Technology

# Educational implementation of SSL/TLS

**Eivind Vinje**

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Danilo Gligoroski, ITEM

Norwegian University of Science and Technology
Department of Telematics

# Problem Description

SSL/TLS is the most common and widely used secure protocol in Internet. It is a package of more than 30 cryptographic primitives and protocols. For students studying information security it is of a crucial importance to have a good understanding of how the different parts are working. The aim of the project will be to develop an educational implementation of SSL/TLS that could be used when teaching information security. The basic idea is to create a simple protocol that encrypts communication between two hosts, implemented in Java. The protocol should include implementations of DH, RSA, AES and SHA-1 accompanied by a graphical interface that will monitor what is happening in every moment of the work of the algorithm (protocol). The graphical user interface should also include statistics on how time consuming each operation has been.

Assignment given: 15. January, 2011
Supervisor: Professor Danilo Gligoroski, ITEM

# Abstract

The Transport Layer Security (TLS) protocol provides secure communication over an untrusted network, such as the Internet. TLS was published as an Internet standard in 1999, closely related to the Secure Socket Layer (SSL) version 3. Today, it is the *de facto* standard for secure end-to-end communication. By combining various cryptographic primitives, TLS provides authentication, confidentiality, integrity, and message forgery detection. The goal when designing the protocol was to create a standard solution for secure communication that was flexible, robust, fast, and easy to implement. TLS has been widely analyzed, resulting in continuous improvement of the protocol and the documentation. This report gives a detailed description of the TLS protocol, and explains why and how it is claimed to be secure. In addition, the report presents the background theory required to understand TLS, such as cryptographic primitives.

In this thesis, we propose a tool, called EduTLS, which can be used when studying information security. The tool is an application developed in Java, and has its own implementation of protocol similar to TLS. The implementation is not compatible with TLS, but has the same structure and architecture. The most complex part of TLS is the Handshake protocol, hence the application has extra focus on this part. The intention with EduTLS is to offer a practical approach when studying the protocol. EduTLS includes implementation of several cryptographic primitives, thus it might also be useful when gaining experience in cryptanalysis.

# Acknowledgment

I wish to thank my supervisor Professor Danilo Gligoroski at the Department of Telematics, Norwegian University of Science and Technology, for giving me opportunity to continue the work from my specialization project.

Gratitude is also given to Mona Nordaune and Professor Danilo Gligoroski for giving me the opportunity to write my thesis in Stavanger instead of in Trondheim. Mona has been more than helpful with the practical issues of writing the thesis from Stavanger. In hindsight I realize it would be more convenient to stay in Trondheim, because some issues are easier to handle face-to-face than over email. However, we do live in a decentralized world, and it has been a great experience to not have the solution in the room next door.

Special thanks are given to Ingrid Strømsheim, who kindly helped proofread this paper and provided valuable feedback.

Finally, I wish to thank my fellow students for the time we have had together, it has been interesting, educational and fun.

Eivind Vinje

# Table of Contents

**Problem Description** i

**Abstract** ii

**Acknowledgment** iii

**List of Figures** vi

**List of Tables** vii

**List of Listings** viii

**List of Abbreviations** x

**Definitions** xi

**1 Introduction** 1
- 1.1 Motivation . . . 2
- 1.2 Limitations . . . 2
- 1.3 Outline . . . 3

**2 Background** 4
- 2.1 History . . . 4
- 2.2 Cryptography . . . 6
  - 2.2.1 Conventional Encryption . . . 6
  - 2.2.2 Hash Function . . . 10
  - 2.2.3 Public-Key Cryptography . . . 12
  - 2.2.4 Secure Random Number Generation . . . 15
  - 2.2.5 Digital Signature . . . 16
  - 2.2.6 Digital Certificate . . . 19
- 2.3 Public Key Infrastructure . . . 21
- 2.4 Software Development . . . 22
  - 2.4.1 Java . . . 22
  - 2.4.2 Automated Testing . . . 22

**3 TLS** 24
- 3.1 Goals . . . 25

3.2 The Record Protocol . . . 26
3.3 The Cipher Change Spec Protocol . . . 28
3.4 The Alert Protocol . . . 28
3.5 The Handshake Protocol . . . 28
3.6 PRF Computation . . . 31

**4 The Implementation** **33**
4.1 Requirements . . . 34
4.2 Overall Approach . . . 35
4.3 The GUI . . . 36
4.3.1 Conceptual View . . . 36
4.3.2 Startup . . . 38
4.3.3 Connect . . . 40
4.3.4 Send Message . . . 45
4.3.5 Performance Test . . . 46
4.4 Classes and Packages . . . 48
4.4.1 The GUI Package . . . 49
4.4.2 The Server Package . . . 50
4.4.3 The Common Package . . . 51
4.4.4 The Crypto Package . . . 52
4.4.5 The TLS Package . . . 56
4.5 The Architecture . . . 59
4.5.1 Applied Patterns . . . 59
4.6 Automated Tests . . . 61

**5 Discussion** **64**
5.1 Security Analysis . . . 65
5.2 Implementation Challenges . . . 66
5.3 Further Improvements . . . 67
5.4 Related Work . . . 68

**6 Conclusion** **69**

**Bibliography** **71**

**Appendix** **77**

**A Tables and Lists** **77**
A.1 The First Fifteen ARPANET Sites . . . 77
A.2 TLS Alert Codes . . . 78
A.3 Modes of Operation . . . 78

**B Source Code** **80**
B.1 Compile and Run the Application . . . 80
B.2 SHA-256 Pseudocode . . . 80
B.3 Calculate SHA-1 and SHA256 Digest . . . 82
B.4 Generating Diffie-Hellman P and G . . . 83
B.5 RSA Key Generation . . . 84

# List of Figures


2.1 Conventional encryption . . . . . 7
2.2 Hash function structure . . . . . 10
2.3 Public-key encryption . . . . . 12
2.4 Digital signature . . . . . 17
2.5 X.509 version 3 certificate structure . . . . . 19

3.1 The SSL stack . . . . . 24
3.2 The Record Protocol . . . . . 27

4.1 Flow chart . . . . . 35
4.2 The GUI - Conceptual view . . . . . 36
4.3 The GUI - Startup . . . . . 38
4.4 The GUI - Add new connection . . . . . 39
4.5 The GUI - Connect . . . . . 40
4.6 The GUI - Send message . . . . . 45
4.7 The GUI - Performance test . . . . . 46
4.8 Packages and classes . . . . . 48
4.9 Model-View-Controller pattern . . . . . 59
4.10 Test coverage . . . . . 61

# List of Tables

3.1 TLS Handshake messages . . . . . 28

4.1 Requirements . . . . . 34
4.2 Source code size summary . . . . . 49
4.3 The GUI package source code . . . . . 49
4.4 The server package source code . . . . . 51
4.5 The common package source code . . . . . 52
4.6 The crypto package source code . . . . . 55
4.7 Connection State . . . . . 57
4.8 The TLS package source code . . . . . 58

A.1 TLS version 1.2 Alert Codes . . . . . 78

# Listings


4.1 Sending ClientHello . . . 40
4.2 Receive ServerHello . . . 41
4.3 Receive Certificate . . . 41
4.4 Receive ServerKeyExchange . . . 42
4.5 Receive ServerHelloDone . . . 42
4.6 Send ClientKeyExchange . . . 42
4.7 Generating Key Block . . . 43
4.8 Send ChangeCipherSpec . . . 43
4.9 Send Finished . . . 44
4.10 Receive ChangeCipherSpec . . . 44
4.11 Finished . . . 44
4.12 Performance Test . . . 46
4.13 IPeerCommunicator Interface . . . 50
4.14 The compareByteArray() Method . . . 52
4.15 The ICipher Interface . . . 53
4.16 The ICompression Interface . . . 53
4.17 The IHash Interface . . . 54
4.18 The IKeyExchange Interface . . . 54
4.19 The IRandomGen Interface . . . 55
4.20 The IHandshakeMessage Interface . . . 57
B.1 Compile and Run EduTLS . . . 80
B.2 SHA Digest Calculation . . . 82
B.3 Generating Diffie-Hellman P and G . . . 83
B.4 Coprime Finder . . . 84
B.5 Modular Multiplicative Inverse Finder . . . 84
B.6 The Miller-Rabin Primality Test . . . 84
B.7 Generating RSA Key-Pair . . . 85

# Abbreviations and Acronyms

**AES** Advanced Encryption Standard

**API** Application Programming Interface

**ARPANET** Advanced Research Projects Agency Network

**ASCII** American Standard Code for Information Interchange

**DES** Data Encryption Standard

**DH** Diffie-Hellman

**DSA** Digital Signature Algorithm

**DSS** Digital Signature Standard

**GUI** Graphical User Interface

**HTTP** Hypertext Transfer Protocol

**HTTPS** Hypertext Transfer Protocol Secure

**IETF** Internet Engineering Task Force

**IP** Internet Protocol

**JVM** Java Virtual Machine

**NIST** National Institute of Standards and Technology

**NTNU** Norwegian University of Science and Technology

**OSI** Open Systems Interconnection

**PKI** Public Key Infrastructure

**PRF** PseudoRandom Function

**RFC** Request For Comments

**RSA** Rivest, Shamir and Adleman

**SHA** Secure Hash Algorithm

**SSL** Secure Socket Layer

**TCP** Transmission Control Protocol

**TLS** Transport Layer Security

**W3C** World Wide Web Consortium

# Definitions

**Authentication** Assurance that an entity is who he/she claims to be.

**Algorithm** A set of rules that specifies how to solve a certain problem.

**API** An interface with rules and specification, which makes it possible for software components to communicate with each other.

**Cipher** An algorithm performing either encryption or decryption.

**Confidentiality** Protection of information against passive attacks, such as eavesdropping.

**Cryptographic primitive** A cryptographic algorithm, such as a hash function or an encryption cipher.

**Digital certificate** An electronic document that combines an identity with a public key.

**Eavesdropping** The act of listening to a private conversation between two parties without their knowledge and consent.

**Git** Version control management system of source code in a development project.

**Github** A web-based hosting service for software development projects that use the Git revision control system.

**Host** A computer connected to a network.

**IEFT** An open community that develop Internet standards.

**Integrity** Protection of information against active attacks, such as tampering.

**JVM** The Java Virtual Machine is a software program used to run Java applications.

**Message forgery** A message sent to deceive the recipient as to whom the real sender is.

**Modulus** The rest from an integer division between two numbers.

**Node** An electronic device attached to the network, such as a router.

**Nonce** A random number that is never reused (number used once).

**OSI model** A standard segmentation of the network protocol into layers.

**Peer** One of the hosts in a point-to-point network link.

**Protocol** A standard, defining a set of rules that determines how data is represented when transmitted over a network connection.

**Public Key Infrastructure** The technical and organizational infrastructure in charge of issuing, distributing, and revoking digital certificates.

**Salt** A random number, or bit sequence, used as input to a hash function in addition to

the message, to avoid dictionary attacks etc.

**Socket** A "communication road" between two processes on different hosts.

**Swing** A Java graphical user interface toolkit.

**W3C** Word Wide Web Consortium: An international community that develop Web standards.

**XOR** A operation that takes two bit patterns and performs a logical XOR operation *(1 XOR 1 = 0, 1 XOR 0 = 1, 0 XOR 1 = 1, 0 XOR 0 = 0)*

**X.509** A public key infrastructure standard

*Note that these definitions are defined according to their role in this report. A more general definition may be more appropriate in other situations.*

Chapter 1

# Introduction

Internet was not designed to be secure; the focus was robustness and scalability. In general, all information sent over a network connection, such as the Internet, can be eavesdropped, tampered, or forged. Netscape Communications was one of the companies that responded to the need for secure network communication, and developed the first version of the Secure Socket Layer (SSL) protocol in 1994. After the success of SSL, it was adopted as an Internet standard by the IETF in 1999 and called Transport Layer Security (TLS). Today, SSL and TLS are the predominant protocols for secure network communication. [1]

Secure Socket Layer (SSL) is a protocol providing end-to-end secure communication. When designing SSL, the goal was a flexible solution where any application could benefit its service. With that in mind, SSL was not incorporated into the Netscape browser, but located at the layer between the browser and the transport layer. From the developer's view, the only modification needed when utilizing SSL was to communicate to the SSL protocol instead of the TCP protocol. This was the thought; to give the developer a unified and simple solution to implement into their applications. The protocol, as the name implies, operates at the socket layer and transports application data from the higher layer protocols, without reflecting about the content. It is intended to use in a client-server model. SSL by itself is useless, but protocols at the application layer, for instance HTTP for web browsing, can benefit the facilities of SSL. [1]

SSL/TLS is the far most popular cryptographic system. Cryptographic systems use cryptographic primitives as building blocks, creating a package that exploits the advantages of each primitive and combines them to cooperate. When studying information security, the focus is often to learn how the cryptographic primitives work, its advantages and disadvantages. This is undoubtedly important, but the primitives by themselves are not secure. For instance, encryption does not ensure integrity. When looking at a cryptographic system, one might get a better overview of how the algorithms can be combined to create the best security possible for a particular situation. [2, 35, 34]

This report and the proposed application are mainly directed towards information security students, who already have good understanding of how computers operate and communicate. To reach out to a broader specter, the theory chapter should provide enough background knowledge that non-security persons can benefit from the project as well. Parts of the theory chapter are cited from the EduSSL[3] report, but most of it has been revised and expanded. Both projects are called "An educational implementation of SSL/TLS", but the application from the specialization project is called EduSSL, while the application presented here is called EduTLS. The reason is simple; EduSSL was based on the SSLv3 specification, and EduTLS is based on the TLSv1.2 specification. Through the rest of this report, the specialization project is referred to as EduSSL, and this project as EduTLS.

The project source code has almost 8000 lines of code, which would result in approximately 200 pages of text, hence it is not included as an appendix. The complete source code is available as an attachment or at the Github page `https://github.com/evinje/EduTLS`. Appendix B.1 lists the commands required to compile and execute the application.

The rest of this report will refer to TLS instead of SSL, but the majority of the protocols are identical.

## 1.1 Motivation

The thesis is a continuation on my specializing project, which was conducted the fall semester of 2010. I wanted to continue the EduSSL project[3] because I felt there was great potential in the idea, and I found the specialization project very interesting. I am fascinated in the SSL/TLS protocol, because it was designed almost two decades ago, and even though the computing power has increased exponential, and used by mobile devices with a fraction of the performance, it is still the dominant end-to-end security standard. There is, however, not too much literature, and especially not with a practical approach, on the subject. Most of the documentation is on how to use existing TLS libraries, without focus on details within the protocol.

The work on my specialization project is similar to the thesis; both propose a tool that can be used when studying information security. In the specialization project, the main focus was to learn and understand the theory, but in addition it proposed the EduSSL application. The EduTLS application presented in this thesis is a complete redesign of EduSSL. The focus has been a more similar architecture to TLS and to make a better presentation of what is happening in the protocol.

## 1.2 Limitations

The scope of this report is not to discuss how security should be considered during a development project, but to emphasize the importance of using the technology as it is intended to be used. Correct use is often related to understanding, and the goal of this report is to give a good understanding of TLS. It is not intended to explain how a native

TLS implementation is accomplished, this can be found in several other projects, see Section 5.4.

Even though some weaknesses in the TLS protocol are mentioned, this report does not include a thorough analysis of its security, and does not explain the weaknesses in detail.

In the EduTLS application, security is not an important issue. The application cannot be used to communicate with other TLS enabled applications, and may contain several vulnerabilities. It should be used for research and educational purposes only.

## 1.3 Outline

The remainder of this report is organized as follows:

**Chapter 2 - Background** gives an overview of the history of Internet and TLS, and presents the theoretical background required to understand TLS. The theory includes the most important cryptographic primitives, with an example of the most popular algorithm of each primitive.

**Chapter 3 - Transport Layer Security** explains the TLS protocol in detail.

**Chapter 4 - The Implementation** starts with the requirements for the application and an explanation of the overall approach, followed by a visual representation of the graphical user interface, an explanation of the classes and packages, the architecture, and finally the automated tests in the project.

**Chapter 5 - Discussion** provides a discussion of the result, challenges, related work, and proposal for future improvements.

**Chapter 6 - Conclusion** presents the conclusion of the thesis.

Chapter 2

# Background

*"Those who do not learn from history are doomed to repeat it"*
- George Santayana

This chapter gives a brief introduction to history and theory relevant to the TLS protocol.

## 2.1 History

This section begins with the history of Internet, and aims to give an understanding of how it has evolved and why it is not secure. This is succeeded by the history of SSL, and the creation of an Internet draft document, which resulted in the standardization of the protocol TLS.

During the Cold War in the 1960s, the US Department of Defense (DoD) created a research project called ARPANET. The goal of the project was to create a reliable and decentralized network to share information between the DoD, the US military, and universities that ran research for them. ARPANET was designed to continue to operate even if one or more of the nodes failed or disconnected, making no single node in control of the network. The ARPANET users comprised a small group of people who generally knew and trusted each other, and of that reason, the concerns were openness and flexibility, not security. In the following decade, other universities and research organizations were connected to the network. Before 1972, fifteen sites were connected into ARPANET, see Appendix A.1 for a complete list. By the end of 1970s, more than two hundred sites were connected, and it had formed a global worldwide network that we today call the Internet. In the 1980s, ARPANET was no longer a defense product, and the commercial Internet Service Providers offered access to the network. Many different projects attempted to create ways to organize the distributed data, for example electronic mail in the 1970s and World Wide Web in the 1990s. Common to them was that security was not considered. [4, 5, 6, 2]

In that time period, Netscape Communications was dominant in the web browser and web server market. They were one of the companies that responded to the need for secure network communication, and developed the Secure Socket Layer (SSL) protocol in 1994. Originally, SSL secured the communication between Netscape's web browser and

server, but the specification stated that the protocol could be used with other applications. Version 1 of SSL was never publicly released, but gained a lot of attention. The year after, Netscape released version 2 of the protocol. The protocol was thoroughly analyzed and many weaknesses were detected. Version 3 aimed to fix the weaknesses and drawbacks, which resulted in a heavily modified protocol, released the year after. [18, 14, 13]

At the same time, there were other companies trying to create their own security solutions. Instead of having several solutions to the same problem, Microsoft and Netscape requested the Internet Engineering Task Force (IETF) to define a standard protocol. The IETF is an open community that develops standards, mostly concerning the technical evolution of the Internet. IETF cooperates with the W3C organization, and together they, among other things, create Internet standards and recommendations. The IETF community is mostly known for its Request For Comments (RFC) documents. An RFC is the official publication channel of a standardization proposal, where a number of rounds of peer reviews are carried through, and may result in an Internet Standard (STD). In 1999, IETF published the first proposal for a new standard; RFC2246[1]. IETF based this standard mainly on the Secure Socket Layer protocol version 3, but in addition, they considered input from several other vendors. The result was the Transport Layer Security (TLS) protocol version 1.0. The RFC states that

> the differences between this protocol and SSL 3.0 are not dramatic, but they are significant enough that TLS 1.0 and SSL 3.0 do not interoperate.

The most important differences are the pseudorandom function, the MAC schemes, alert codes, supported and mandatory cipher suites, and the calculation of master secret. [10, 14, 13]

After seven years, RFC4346 was released, with the definition of TLS version 1.1. The difference between version 1.0 and 1.1 are not significant, but protection against the discovered attacks was added, and support for new cryptographic algorithms was defined. The fact that TLS version 1.0 was not upgraded for seven years, confirmed how well the protocol was designed, considering the explosion of its popularity. In 2008, TLS version 1.2 was defined. This version is currently the latest release. The most important improvements are discussed later in this chapter.[1, 4, 5, 11, 12, 10]

[1]Each RFC is assigned a unique number, for TLS version 1.0, it is 2246

## 2.2 Cryptography

*"Most security failures in its area of interest are due to failures in implementation, not failure in algorithms or protocols"*
- The National Security Agency

To understand how TLS works, it is important with basic knowledge of cryptography. Cryptography is a subfield within cryptology[2]. According to William Stallings[7],

> cryptology is the study of techniques for ensuring the secrecy and/or authenticity of information.

Cryptanalysis is another subfield within cryptology. Cryptography is the study of how to design functions, whereas cryptanalysis tries to find vulnerabilities in the design. Cryptanalysis is not discussed in this report.

The topics in cryptography relevant to this project are encryption algorithms, hash functions, secure random generator, digital signatures, and digital certificates. Encryption algorithms are divided into two categories; conventional encryption and public-key cryptography. Encryption is the process of converting readable information into something that is not understandable without knowing a secret piece of information, usually referred to as the key.

### 2.2.1 Conventional Encryption

*"Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break"*
- Bruce Schneier

In conventional encryption, the same key is used for both encryption and decryption. The concept has existed for more than 2000 years. One of the first, and maybe the most known, is the Caesar cipher. In the Caesar cipher, each letter in the message is replaced with the letter located three positions later in the alphabet. This means that A becomes D, D becomes G and Z becomes C. The key in this cipher is the knowledge of swapping the letters three positions in the alphabet. [7, 2]

[2]Cryptology derives from the Greek *kryptós lógos*, meaning hidden word[15]

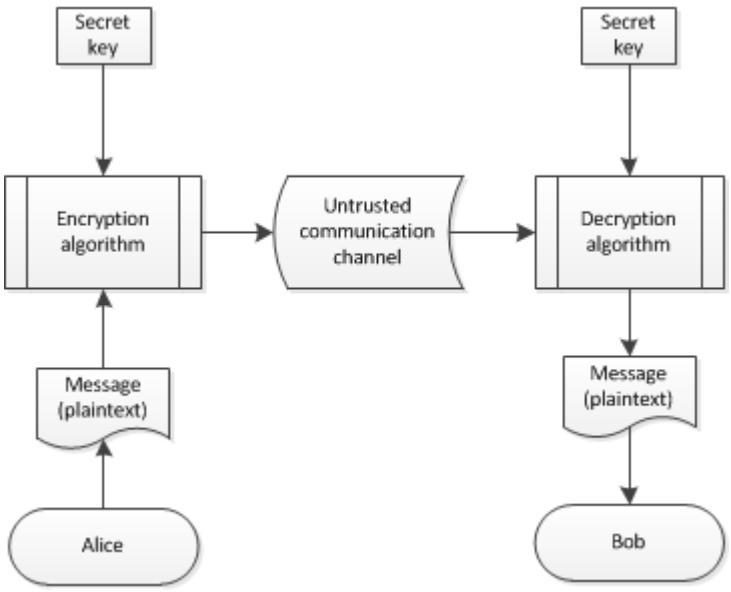


Figure 2.1: Conventional encryption

> Assume two parties, Alice and Bob, are going to exchange a message secretly. Alice encrypts the message with an encryption algorithm and a secret key, and transmits the message to Bob over an untrusted communication channel. Bob receives the message, and decrypts the message with the corresponding decryption algorithm and the same key.

Conventional encryption is also called symmetric key encryption, private key encryption, or secret key encryption. All three latter names are derived from the fact that both parties, the sender and the receiver, must share a secret key that is known only to them. The encryption and decryption is performed by using an algorithm, called a cipher. The encryption and decryption cipher may be identical, but that is not always the case[3]. In addition to sharing a secret key, both parties must know which cipher they will be using. The cipher does not need to be kept secret; it is the key that protects the message. When encryption is performed, the cipher takes a message, called the plaintext, and the key, and transforms it to an encrypted message. The encrypted message is referred to as the ciphertext. To recover the plaintext, the decryption cipher is used. By giving the decryption cipher the same key and the ciphertext, it transforms back the original plaintext. [7, 2]

To summarize, the following terminology is used:

**Plaintext** is the original message to be sent

**Cipher** is the algorithm used for the transformation. Can be divided into encryption cipher and decryption cipher

**Key** is the private key used by the cipher to transform the plaintext

**Ciphertext** is the scrambled/encrypted message, the output from the cipher

[3]In the Rijndael cipher, the encryption and decryption cipher is different, but in the DES cipher, they are identical

Most of today's conventional encryption ciphers use a combination of substitution and permutation. In substitution, each unit is replaced with another unit. The substitution can be a lookup from a predefined table or according to an algorithm. It is important though, that the process can be reversed, and that no replacement is ambiguous. Permutation is a method where the units are reordered. All the units will remain, but the position has changed. The conventional cipher used in TLS is either a stream cipher or a block cipher[4].

A block cipher divides the plaintext into fixed-size blocks, and encrypts one block at a time into a block of ciphertext with the same size. Block ciphers may have several *modes of operation*. The mode of operation solves the problem that a block cipher is designed to encrypt only one single data block, while in practice the message is of variable-length and usually larger than the block size. The Electronic Code Book mode (ECB) is the simplest, where each block is encrypted separately with the same key. The disadvantage is that identical plaintext blocks results in identical ciphertext blocks, thus it does not hide patterns. The Cipherblock Chaining mode (CBC) solves the problem with ECB by XORing the plaintext block with the previous ciphertext block. The first plaintext block is XORed with an initialization vector (IV), that must be kept as secret. NIST has specified a total of nine modes of operation, listed in Appendix A.3. [16]

Stream cipher combines the plaintext with a key stream and typically XOR these, one unit at a time. The challenge is to create a potentially infinite key stream, which is usually solved by a *feedback shift register* (FSR). The FSR has a register, a feedback function, and an internal clock. By using the clock, the encryption transformation varies with time. EduTLS only support block ciphers, thus the stream cipher is not explained in more detail.[2, 15]

Conventional encryption has the benefit of being very fast, but has the major disadvantage of key distribution; the parties must have exchanged the secret key in advance of the communication.

#### 2.2.1.1 AES

The Advanced Encryption Standard (AES) algorithm is originally named Rijndael. Between 1997 and 2000, NIST arranged an open competition of becoming the next conventional encryption standard, a successor for the current encryption standard DES. Rijndael won this competition in October 2000, and has since commonly been referred to as AES[5]. The Rijndael algorithm was designed by Vincent Rijmen and Joan Daemen.[15]

This chapter gives a brief overview on how the AES algorithm operates. AES is a block cipher, where the block size is 128 bits, and the key size is 128, 192 or 256 bits. Depending on the key size, the algorithm performs a number of repetitions, or rounds[6], that converts the plaintext into ciphertext. AES operates on a two-dimensional matrix of bytes, called the *state*. One state contains one block, and the matrix is $4\text{x}N$, where $N$ is the block size divided by 32. Since the current AES specification only adopted one block size, the state

[4] Several other types exist, such as monoalphabetic, polyalphabetic, and steganographic
[5] AES is not precisely Rijndael, because Rijndael supports a larger range of block sizes and key lengths
[6] 10, 12 or 14 rounds for 128, 192 or 256 bit keys, respectively

matrix is always 4x4 bytes[7]. [15]

AES performs most of its calculations within the finite field. A finite field is a set of elements, where the number of elements can be represented by a natural number. All operation on those elements will result in a new element contained in the same field, and each operation has its inverse operation. [2, 15]

The AES encryption algorithm[15];

$KeyExpansion()$
$s \longleftarrow input$
$s \longleftarrow AddRoundKey()$
$loop from\ R \longleftarrow 1\ to\ (NumberOfRounds - 1)$
$s \longleftarrow SubBytes()$
$s \longleftarrow ShiftRows()$
$s \longleftarrow MixColumns()$
$s \longleftarrow AddRoundKey()$
$end\ loop$
$s \longleftarrow SubBytes()$
$s \longleftarrow ShiftRows()$
$s \longleftarrow AddRoundKey()$
$output \longleftarrow s$

The first step in the cipher is the key expansion function, where the Rijndael key schedule creates one key for each of the rounds, derived from the main key. Before the rounds start, the input bytes are copied into the state, denoted as $s$, followed by an initial *AddRoundKey* transformation. As shown in the algorithm, all the rounds are identical, except for the last one. [15]

A short description of the encryption steps;

**AddRoundKey** performs a bitwise XOR between the state and the round key.

**SubBytes** performs substitution according to a table, called the S-box.[8]

**ShiftRows** cyclically shift the bytes in every row to left, where the number of places to shift differs for each row.

**MixColumns** multiplies each column of the state with a fixed polynomial

The decryption is accomplished by doing the same steps, but with the corresponding reverse function. For example, *SubBytes* in decryption becomes *InvSubBytes*, where the S-box is the inverted substitution box.

[7] 4x4 is 16 bytes, which is 128 bits.
[8] The Substitution box in AES is a 16x16 matrix with predefined values

### 2.2.2 Hash Function

*"There are, in fact, no known instances of functions which are provably one-way with no assumptions."*
— Alfred J. Menezes

A hash function, one-way function, or message digest function, is a procedure, which takes a variable length input and compresses it to a fixed size output. This means a single character will produce an equal size output as a large document. The output is referred to as the hash value, digest, checksum or fingerprint. The same input will always result in the same hash value, and the process is not reversible. Hash functions are commonly used to compare values. It can, for instance, ensure that a message has not been tampered with, by comparing the original hash value with the current hash value.[15, 2, 17]

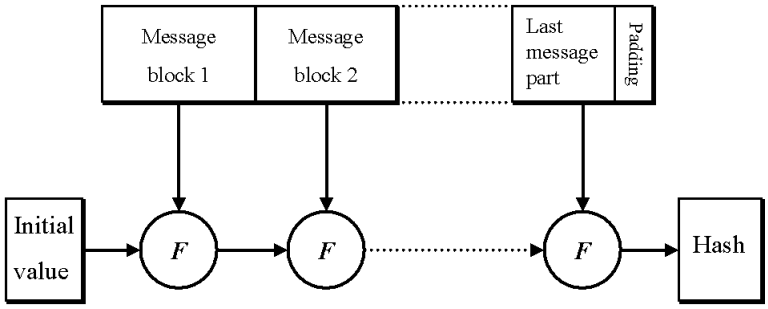


Figure 2.2: Hash function structure

The figure above illustrates the typical structure of a hash function. F denotes a compression function, which takes a fixed-size input and turns it into a shorter, fixed-size output. The compression function is repeated a number of times until the entire message have been processed. The initial value is specific for each algorithm. This structure is called the Merkle-Damgård construction, after its inventors Ralph C. Merkle and Ivan B. Damgård. [15, 2, 17]

A cryptographic hash function is a hash function that satisfy the following properties:[7, 15, 17]

**Easy to compute** For any given input $x$ it is easy to compute the hash value $h$

**Preimage resistance** For any given hash value $h$ it is impossible[9] to find a corresponding input $x$

**Second preimage resistance** For any given message $x$ it is impossible[9] to find another message $y$ such as $H(x)=H(y)$ and $x \neq y$

**Collision resistance** It is impossible[9] to find two different messages $x$ and $y$ such as $H(x)=H(y)$ and $x \neq y$

[9]Impossible in this context means computational infeasible

The hash function is noted as $H$, input message $x$ and the resulting hash value $h$; $H(x)=h$.

The National Institute of Standards and Technology has approved five cryptographic hash functions; SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512. [39]

#### 2.2.2.1 SHA-256

The Secure Hash Algorithm (SHA) is a set of cryptographic hash functions, published by NIST. SHA-0 was published in 1993, but due to some early discovered weaknesses[36], the algorithm was corrected and released two years after by the name SHA-1. In 2001, NIST published the second generation of SHA, SHA-2, which is a family of four different hash functions. The SHA-2 family consists of SHA-224, SHA-256, SHA-384 and SHA-512, where the number after SHA is the digest length in bits. There are only minor differences in the algorithms; the initializing variables and number of rounds are most important.[37]

In 2007, NIST announced an open competition to the next standard in the SHA family. In 2012, the winner is chosen and the winning algorithm will be named SHA-3.

SHA-256 takes a message input of arbitrary length, max $2^{64} - 1$ bits, and outputs a 256 bits message digest. The algorithm uses 64 round constants and 8 working variables. The round constants are initiated from the first 64 primes[10], and the working variables are initiated from the first 8 primes[11]. After the initiation, it performs 64 rounds of operations on each 512 bits chunk. Each chunk it split into sixteen 32 bit words and each round consist of a series of bit operations, additions, and shift operations on those words. The operations can be summarized as bitwise AND, OR, XOR and complement operations, left-shift and right-shift, and addition in modulo $2^{64}$. After the final round on the last chunk, the eight working variables are concatenated, resulting in the 256 bit hash value. See Appendix B.2 for a complete pseudo-code of SHA-256. [37]

[10] The first 32 bits of the fractional parts of the cube roots of the first 64 primes
[11] The first 32 bits of the fractional parts of the square roots of the first 8 primes

### 2.2.3 Public-Key Cryptography

*"The obvious mathematical breakthrough would be development of an easy way to factor large prime numbers"*
- Bill Gates

The public-key cryptography scheme was first introduced in 1976 by Whitfield Diffie and Martin Hellman in the paper *New Directions in Cryptography*. Diffie and Hellman's public-key concept was the most spectacular development in the history of cryptography, which dates back almost 4000 years when the Egyptians used the hieroglyphs[12].[2, 8]

Public-key cryptography, or asymmetric key algorithms, is the common denomination for an encryption system where the entities do not share a secret key. Instead of one key, there is a key-pair, where one key is private and the other key is public. The key-pair has properties such as when using one of the keys to encrypt a message, only the other key of that key-pair is able to decrypt the message. This revolutionized how cryptographic systems could be designed. [15]

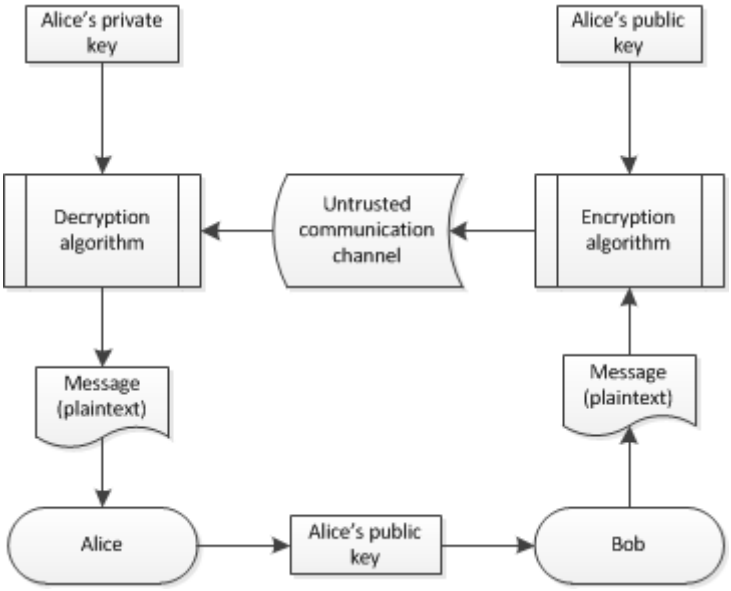


Figure 2.3: Public-key encryption

Assume two parties, Alice and Bob, are going to exchange a secret over an unsecured network link. Alice has her own key-pair, and sends the public key to Bob. With Alice's public key, Bob encrypts the secret message, and transmits the encrypted message to Alice. With her private key, Alice decrypts the message.

[12]The hieroglyphs is the earliest known cryptography, made by the Egyptians around 1900 BC

Not all public-key schemes offer encryption. Public-key schemes can be categorized as follows:

**Encryption;** the key-pair is used for encryption and decryption of a message, as demonstrated in Figure 2.2.3.

**Key distribution;** a new key is exchanged with the use of public-key cryptography, and this key is used in a conventional encryption algorithm. This topic is described more in Section 2.3

**Digital signature;** a cryptographic hash function is used to create fingerprint of a message, and the private key is used to encrypt the fingerprint to create a signature. This topic is described more in Section 2.2.5

In public-key cryptography, the algorithms for encryption and decryption are mathematical functions. These functions often include a combination of exponential and modulus computation. The keys are large numbers, and generated with the use of an algorithm. When having one of the keys, it is impossible[13] to discover the paired key.

Public-key encryption is, however, not the solution to every problem. Compared to conventional encryption, this scheme involves very time-consuming computation, making it unpractical when encrypting large amounts of information. A 128-bit key in conventional encryption has equal computational security as a 3072-bit key in public-key encryption. According to several resources[28, 29, 30], conventional encryption is about 1000 times faster than public-key encryption. Hence, the public-key encryption is most often used to distribute keys for symmetrical encryption or to create digital signatures. [7, 22, 27]

#### 2.2.3.1 RSA

The first public-key algorithm was published in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman; the RSA algorithm. The RSA algorithm can be used for both encryption, key exchange and signature. This chapter demonstrates how the RSA algorithm operates by showing a practical example of the computations.

A RSA system uses the following terminology

The private key is the couple (n, d)

The public key is the couple (n, e)

where

$n$ is the modulus: the product of two large primes p and q

$d$ is the private exponent

$e$ is the public exponent

[13] Impossible in this context means computational infeasible

### Key Generation

Before the RSA system can be employed, the key-pair must be generated. The first step is to randomly generate two prime numbers with approximately the same size. Compute the modulus $n$ by multiplying the primes $p$ and $q$;

p = 23
q = 29
n = p*q = 23*29 = 667
$\phi(n)$[14] $= (p-1)*(q-1) = 22*28 = 616$

The next step in the generation process is to randomly select an integer $e$ with a size between 3 and $\phi(n)$, and $e$ is a coprime[15] of $\phi(n)$;

e = 17

There are many techniques to generate $e$, a Java example of how to find coprimes of a number is given in Appendix B.5.

The last step in the generation is to determine the private exponent $d$, which must satisfy the following equation

$d * e = 1 \ mod \ \phi(n)$
$d = e^{-1} \ mod \ \phi(n)$
$d = 17^{-1} \ mod \ 616$
$d = 145$

A Java example of how to solve the equation above is given in Appendix B.5.

The public key consists of n and e (667, 17), and the private key is n and d (667,145). The key-pair owner must share the public key (667, 17) with the communicating party

### Encryption and Decryption

To encrypt a message $m$, the message must be represented as a number, and must not be larger than $n$. To encrypt the message $m = 71$ with the public key, compute the ciphertext;

$c = m^e \ mod \ n$
$c = 71^{17} mod \ 667$
$c = 225$

To decrypt the message, the private key is used

$m = c^d \ mod \ n$
$m = 225^{145} \ mod \ 667$
$m = 71$

This process can be performed in the opposite direction; the message is encrypted with the private key, and decrypted with the public key.

[14] $\phi$ is the Eulers totient function, and $\phi(n)$ is the number of coprimes of $n$ between 1 and $n$-1
[15] Coprimes are two integers that have no common divisor other than 1

### 2.2.4 Secure Random Number Generation

*"Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin."*
– John von Neumann

Random numbers are often a vital part of a cryptography system. In cryptography, it is commonly referred to as a pseudo-random number generator (PRNG) or a pseudo-random bit generator (PRBG), where the latter generate a potentially infinite bit sequence. The first definition of a PRNG was created by Manuel Blum and Silvio Micali in the early 1980s[15], where they stated that the random generator is cryptographically secure if one cannot guess the next number in a sequence, with knowledge of prior numbers, better than guessing at random. In cryptography, random numbers are used for, among other areas, key generation, one-time pads, nonce, and salt. Examples are the $p$ and $q$ primes in RSA, and the secret key in AES.

A true random number generator can only come from a naturally occurring source. Creating hardware or software to exploit randomness from a physical means is unpractical because it tend to be either too costly or too slow. In addition, the generator source must not be able to observe or manipulate by an adversary. [2]

#### 2.2.4.1 Blum-Blum-Shub PRBG

The Blum-Blum-Shub pseudo-random bit generator is a cryptographically secure bit generator under the assumption that integer factorization is intractable. The algorithm demonstrated below generates a sequence of pseudo-random bits of length $l$. [2]

Generate two large primes $p$ and $q$, where both are congruent to *3 modulo 4*. These must be kept secret.
Compute $n = p*q$
Select a random seed $s$, where s is between 1 and $n$-1 and $s$ and $n$ are coprimes
Compute $x = s^2 \ mod \ n$
For $i = 1$ to $l$
$x_i = x_{i-1}^2 \ mod \ n$
$z_i = the \ rightmost \ bit \ of \ x_i \ (the \ least \ significant \ bit)$

The bit sequence $z_1, z_2, z_3, ..., z_l$ is the output of $l$ pseudo-random bits

### 2.2.5 Digital Signature

*"When it comes to privacy and accountability, people always demand the former for themselves and the latter for everyone else."*
— David Brin

A digital signature is a counterpart to a handwritten signature, and attached to a document as a proof of authorship, or as an agreement with the content. [2, 42]

The National Institute of Standards and Technology (NIST) defines a digital signature as[40]

> the result of a cryptographic transformation of data that, when properly implemented, provides a mechanism for verifying origin authentication, data integrity and signatory non-repudiation.

A signature must meet the following criteria[2, 42]:

**Authentic;** a signature must be a proof that the signer intentionally signed the document

**Unforgeable;** a signature is a proof that it was the signer, and no one else, that signed the document

**Not reusable;** a signature is a part of the document and can not be used on any other document

**Unalterable;** a signature must proof that the document has not been altered after the signature was created

**Repudiated;** a signer cannot deny he signed the document

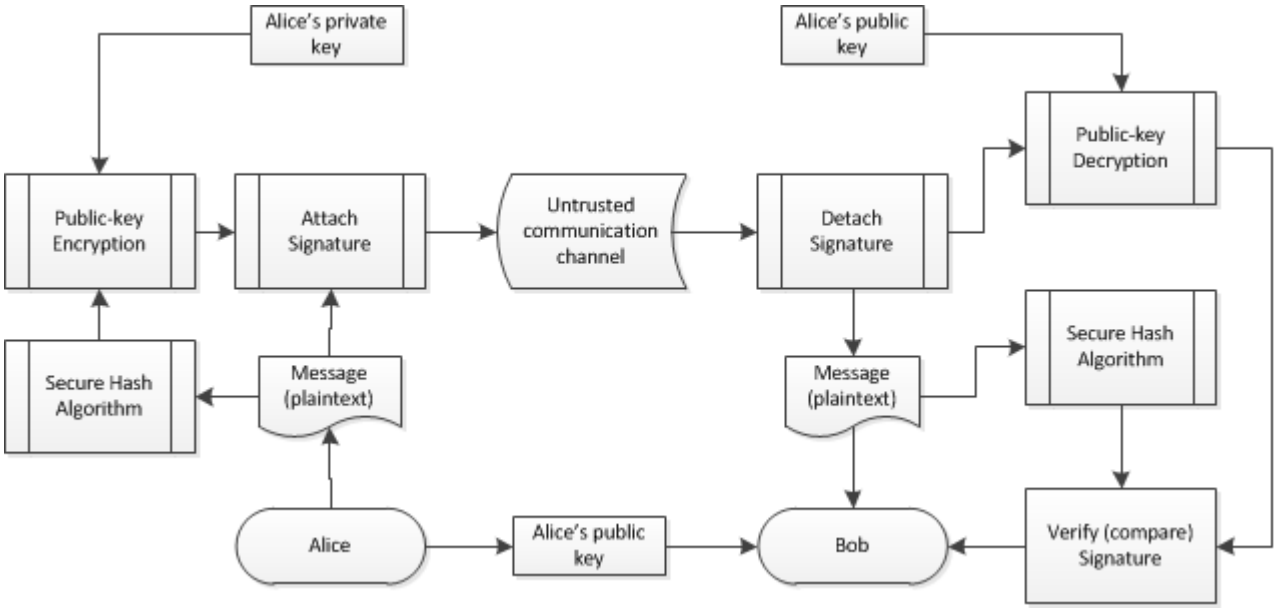


Figure 2.4: Digital signature

> Assume two parties, Alice and Bob, are going to exchange a message over an unsecured network link. The content of the message is not sensitive, but it is important that it is not altered during transfer. Alice has her own key-pair, and sends the public key to Bob. Before Alice sends the message, she calculates the message's fingerprint with a secure hash algorithm. She then creates a signature by encrypting this fingerprint with a public-key algorithm and her private key. This signature is attached to the message before sending it. When Bob receives the message with the signature, he detaches the signature, decrypts it with Alice's public key to obtain Alice's fingerprint. Bob then calculates the fingerprint of the message with the same secure hash algorithm, and compares this value with the decrypted value that Alice sent. If these are equal, the message has not been altered and Bob knows it is the same message that Alice sent.

According to the Digital Signature Standard (DSS), a digital signature is a combination of a secure hash function and public-key cryptography. NIST has specified three approved standards for digital signature; the Digital Signature Algorithm (DSA), the Elliptic Curve Digital Signature Algorithm (ECDSA) and Rivest Shamir Adleman (RSA) algorithm. All three algorithms are used in conjunction with an approved hash function that is listed under Section 2.2.2.[40]

#### 2.2.5.1 DSA

The Digital Secure Algorithm (DSA) consists of a key-pair, a per-message secret number, an approved hash function, and the message to be signed. The DSA specification emphasizes that the key-pair shall only be used for a fixed period of time before generating a new key-pair. The public key however, must be kept in order to verify old messages. [40]

The DSA algorithm consist of three phases; key generation, signing, and verification. The steps in the phases are according to the NIST specification[40, 41].

### Key Generation

In the key generation process, an approved hash algorithm must first be chosen. The next step is to decide key length $L$ and $N$, where NIST recommends[40] key pairs of (1024,160), (2048,224), (2048,256), and (3072,256) bits. It is important that $N$ is equal to, or less than, the size of the hash digest.

Calculation of the key-pair:
Generate prime $q$ of $N$ bits
Generate prime $p$, such that p - 1 is a multiple of $q$, and $p$ is $L$ bits
Calculate $g$, such that $g = h^{(p-1)/q} \ mod \ p$ and h is a random number less than $p$-1.
Generate a random $x$ that is positive and less than $q$
Calculate $y = g^x \ mod \ p$

The public key is the parameter (p, q, g, y) and the private key is (x).

### Signing

To sign a message, a random per-message value $k$ is generated. The $k$ parameter must be less than $q$.
To create the signature, calculate
$r = (g^k \ mod \ p) \ mod \ q$
$s = (k^{-1} \ (hash(message) \ + \ x * r)) \ mod \ q$
The signature of message is (r,s). The + sign denotes concatenation.

### Verification

The signature verification process may be done by the recipient or a third-party. To perform the verification, one must have the public key (p, q, g, y) of the signer.

To verify the message, calculate
$w = s^{-1} \ mod \ q$
$u1 = (hash(message) * w) \ mod \ q$
$u2 = (r * w) \ mod \ q$
$v = ((g^{u1} * y^{u2}) \ mod \ p) \ mod \ q$

The signature is valid if v = r.

The Digital Signature Standard is not a part of TLS, but the signing principle is a crucial part of any cryptographic system. TLS has its own signing method, described in Section 3.2.

### 2.2.6 Digital Certificate

*"Security is always excessive until it's not enough."*
— Robbie Sinclair

A digital certificate, or public-key certificate, is an electronic document that binds an identity to a public key. In other words, a digital certificate is an authentication document; it attests the ownership of a public key. To have trust in the certificate, it is signed by a trusted third party, called a certificate authority[16] (CA). The most common standard for digital certificates is the X.509 format.

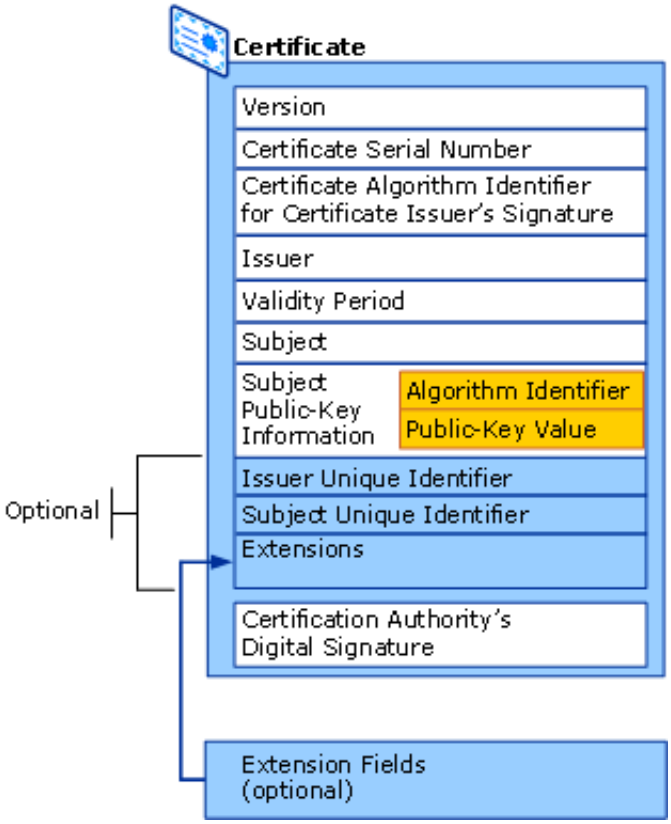


Figure 2.5: X.509 version 3 certificate structure

A X.509 certificate includes the following elements:[7, 19, 23, 24]

**Version** The X.509 version which the certificate conforms.

**Certificate Serial Number** A unique certificate identifier within the issuing Certificate Authority.

**Certificate Algorithm Identifier** The algorithm used to sign the certificate.

**Issuer** The name of the Certificate Authority who issued the certificate.

**Validity period** The period of time where the certificate is valid. It contains both a start date and an expiration date.

[16] This is a part of a public key infrastructure, explained in the next section

**Subject** The name of the entity the certificate belongs to.

**Subject Public-Key Information** The public key belonging to the entity of the Subject and the algorithm associated with the public key.

**Issuer Unique Identifier** Information that can be used to uniquely identify the issuer of the digital certificate.

**Subject Unique Identifier** Information that can be used to uniquely identify the owner of the digital certificate.

**Extensions** Additional information that is related to the use and handling of the certificate.

**Certificate Authority Digital Signature** Consists of a hash value of the certificate encrypted with the Certificate Authority's private key.

There are currently three versions of the X.509 certificate. The *Issuer Unique Identifier* and *Subject Unique Identifier* were added in version 2, and *Extensions* added in version 3.

## 2.3 Public Key Infrastructure

*"In theory, one can build provably secure systems. In theory, theory can be applied to practice but in practice, it can't."*
— M. Dacier, Eurecom Institute

> "A public key infrastructure (PKI) binds public keys to entities, enables other entities to verify public key bindings, and provides the services needed for ongoing management of keys in a distributed system"[43]

This includes, but is not limited to, an architecture to issue, distribute, validate, and revoke digital certificates. It is important to understand that PKI is not a technology, but a description of how organize a public-key cryptography infrastructure. The Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T) has created a standard, X.509, which is the recommended PKI standard for public key certificates. [7, 25, 43]

A PKI consists of several entities/elements:[2, 7, 43]

**End user** The entity that consumes the PKI service or an entity that can be identified in a digital certificate

**Certification authority** A trusted third party (an organization) which creates and sign certificates. It is usually formed in a hierarchy, and the digital certificate may have a chain of several Certificate Authorities in the signature. The CA is the basic building block in a PKI.

**Registration authority** An optional entity, which can be delegated a number of administrative tasks by the Certificate Authority

**Revocation authority** An optional entity, which can be delegated to publish Certificate Revocation Lists by the Certificate Authority

There is no predefined list of certificate authorities; this is up to each system to provide. When an entity requests a signature from a CA, the CA must verify the entity's identity. In some cases this is an email or a phone call, but in a more strict policy the person must meet up himself/herself with an identification.

## 2.4 Software Development

This section explains briefly what Java and Automated Testing is.

### 2.4.1 Java

Java is an object-oriented programming language, released in 1995 by Sun Microsystems. The syntax is inspired by the popular programming languages C and C++, but has a simpler object model. It is widely used and taught at many universities, including NTNU. There are many advantages when choosing Java as the programming language. For this project, the vital ones are its cross-platform capability[17], the diversity of discussion forums and on-line documentation, and the single most important one; when creating an application that is going to be used when teaching, the selected programming language must be known by the students. The fact that Java is a high-level language is another good argument for choosing it as the programming language in this project. When one uses the SSL/TLS capabilities in Java, the usage is almost identical to regular socket communication. For the developer, this can often result in a situation where he *know how to use it, but do not understand how it works*. Security is more than just using strong keys and well-tested cryptographic algorithms, it is an aspect that must be considered during the whole process, and the implementation must be done with care. [9]

There are two terms in object-oriented programming that is required to know when reading this report; class and interface. A class is a prototype/blueprint of an object. The most important content of a class is variables and methods. When an instance of a class is created, it called an object. An interface defines an abstract class, where methods with no content are defined. A class can "implement" the interface and create all the methods defined in the interface. The interface can be seen as a type definition, where every class implementing that interface solves the same issue. The advantage with the interface concept is the loose coupling between the classes, making it easier to change certain parts of an application.

### 2.4.2 Automated Testing

Automated testing is the concept of writing software tests that is used to confirm correctness or detect bugs in a development project. Creating the tests is mostly a manual process, but when a test is created, it can be run as many times as desired. It is the most popular testing method today, but is not sufficient by itself, but a complementary to other testing methods. [44]

Automatic tests are categorized as following: [46]

**Unit tests** isolate a specific component and test only this component without dependencies to other parts of the system. If it relies on other components it is common to create mock objects that simulate expected behavior of the other component. Unit

[17] It can be used in all the major operating systems, like Windows, Linux and Mac

tests are efficient because it does not rely on the environment or other functionality that may introduce bugs to the specific component.

**Integration tests** confirm that a component, or a set of components, interact with rest of the system as expected. The integration tests run after unit tests to avoid that component bugs cause the tests to fail; it is the interaction that shall be verified.

**Functional tests** , or system tests, is black-box testing where implementation details like source code or design logic are unknown. The system is seen from a user point of view, without considering platform, programming language, or any other technical details. Functional tests shall verify the behavior of the system as a whole.

The automated tests only confirm correct behavior for the scenarios the tests asserts, thus the quality of the tests depend largely on how efficient the tests are. [44, 46]

Chapter 3

# Transport Layer Security

The Transport Layer Security (TLS) protocol offers secure communication over an untrusted network connection; it provides authentication, confidentiality, integrity and message forgery detection. Confidentiality is the concept of ensuring that information is accessible only to those authorized to have access. In network communication, this is the process of preventing eavesdropping. The communication can still be eavesdropped, but TLS ensures that only the authorized parts are able to understand the content, by using encryption. [10]

While confidentiality ensures that an unauthorized party cannot read the content, preserving integrity is the process of ensuring that the information is not changed by anyone not authorized to do so. Integrity is preserved by including a message authentication code (MAC). When a message is received, its secure hash value is generated by using a secret key, known only to the authorized parties, and the content of the message. If this value is equal to the one attached to the message, it has not been modified since the creation of the message. Message forgery detection is the process of ensuring that a message is sent from whom it is claimed to be sent from. A common message forgery scenario is that a message has been eavesdropped and resent later. To prevent this, TLS uses sequence numbers on the messages. The sequence number is not a part of the message, but is included when calculating the secure hash value, thus a message cannot be sent more than once. TLS offers authentication by using digital certificates. [31, 14, 13, 10]

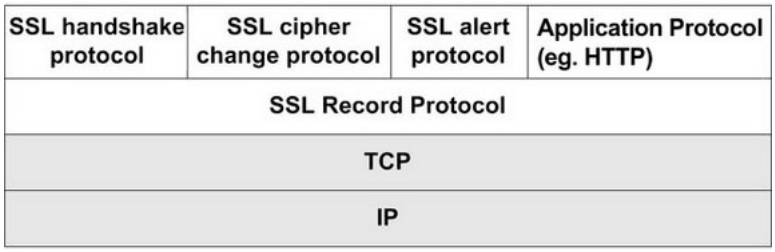


Figure 3.1: The SSL stack

In the OSI model, TLS is above the transport layer and below the application layer. Wikipedia defines a layer as

a collection of conceptually similar functions that provide services to the layer above it and receives services from the layer below it. [26].

TLS relies on the TCP protocol at the transport layer, which offers a reliable end-to-end service. TLS is not just one single protocol, but four protocols, organized in layers; the Handshake protocol layer and the Record protocol layer. The Record protocol is at the Record protocol layer, and the Handshake protocol layer consists of the Handshake protocol, the Alert protocol, and the Change Cipher Spec protocol. Each of these protocols is explained later in this section. [11, 1, 7]

At the lowest layer of TLS, the Record protocol provides security to the higher-layer protocols. The Handshake protocol, the Alert protocol, and the Change Cipher Spec protocol are management protocols. If an application transmits a package to a destination that is not associated with a valid session, the Handshake protocol is used to create a session. A session contains a session identifier, the peer certificate, compression method, cipher suite, master secret, and a flag indicating whether the session can be resumed. A cipher suite is a specified combination of a key exchange algorithm, a conventional encryption algorithm, and a MAC algorithm. [11]

TLS is a hybrid system, combining a public-key scheme and a conventional encryption scheme. The public-key algorithm is used to exchange key, and the conventional encryption algorithm is used to encapsulate the data traffic with a secret key calculated from the key exchange. This approach exploits the best from both schemes, the convenience of public-key and the efficiency of conventional encryption.

When a TLS connection is established, five general phases are executed [1]:

**Phase 1** Create a TCP connection by performing a three-way-handshake

**Phase 2** Perform a TLS handshake, either a full or a session resume

**Phase 3** Transfer the application data, encrypted with the keys exchanged during the handshake

**Phase 4** Close the TLS connection

**Phase 5** Close the TCP connection

## 3.1 Goals

The goals of the TLS protocol, cited from the RFC5246[10], in their order of priority:

1. **Cryptographic security:** TLS should be used to establish a secure connection between two parties.

2. **Interoperability:** Independent programmers should be able to develop applications utilizing TLS that can successfully exchange cryptographic parameters without knowledge of one another's code.

3. **Extensibility:** TLS seeks to provide a framework into which new public key and bulk encryption methods can be incorporated as necessary. This will also accomplish two sub-goals: preventing the need to create a new protocol (and risking the introduction of possible new weaknesses) and avoiding the need to implement an entire new security library.

4. **Relative efficiency:** Cryptographic operations tend to be highly CPU intensive, particularly public key operations. For this reason, the TLS protocol has incorporated an optional session caching scheme to reduce the number of connections that need to be established from scratch. Additionally, care has been taken to reduce network activity.

## 3.2 The Record Protocol

The Record protocol receives data from higher layers, such as the HTTP application protocol, and transmits it to the TCP layer after the processing is finished. The process involves fragmenting the data, compressing, adding MAC and perform encryption, and append header information. When a host receives incoming TLS packets, the TCP layer transmit the packet to the Record protocol, where the same procedures are performed in reverse order, and finally delivered to the appropriate protocol at the layer above. The role of the Record protocol is to offer integrity and confidentiality to the upper layer protocols. [1, 10]

The first step, fragmentation, consists of splitting the received data into TLSPlaintext record blocks/chunks that are $2^{14}$ bytes or less. The TLSPlaintext blocks will be compressed with the compression algorithm defined in the session state. Initially, this compression algorithm is CompressionMethod.null, but the handshake may change this if both parties support a mutual, lossless compression algorithm. After the compression, the TLSPlaintext is transformed to a new structure referred to as TLSCompressed. The block is called TLSCompressed even if the compression algorithm is empty and no compression is performed. [10]

To ensure message integrity, a message authentication code (MAC) is attached to the TLSCompressed block. The MAC is generated by using the HMAC_hash function, where hash is the MAC algorithm specified in the chosen cipher suite of the current session. The HMAC_hash function takes a secret and a seed as input, and produces a fixed-size output. The secret used is either *client_write_MAC_secret* or *server_write_MAC_secret*, depending who the sender is. The seed is a concatenation of the sequence number, content type, protocol version, the length of TLSCompressed, and the content of TLSCompressed. The HMAC_hash function is defined in Section 3.6.

The next step is to transform the TLSCompressed, and its attached MAC, into a TLSCiphertext structure. If the current session state has no cipher algorithm specified[1], the block will remain unchanged. The secret used in this operation is either *client_write_key* or *server_write_key*, depending on who the sender is. [10]

[1] i.e. BulkCipherAlgorithm.null

Finally, before transmitting the result to the underlying protocol, the Record protocol header is appended. The header consists of content type, protocol version and length of the TLSCiphertext block. Content type specifies the higher-level protocol that delivered the data to the Record protocol, protocol version specifies the SSL/TLS version being employed, length is the size of the data fragment, and data is the content of the application data fragment. [11]

The procedures described above are summarized in the figure below.

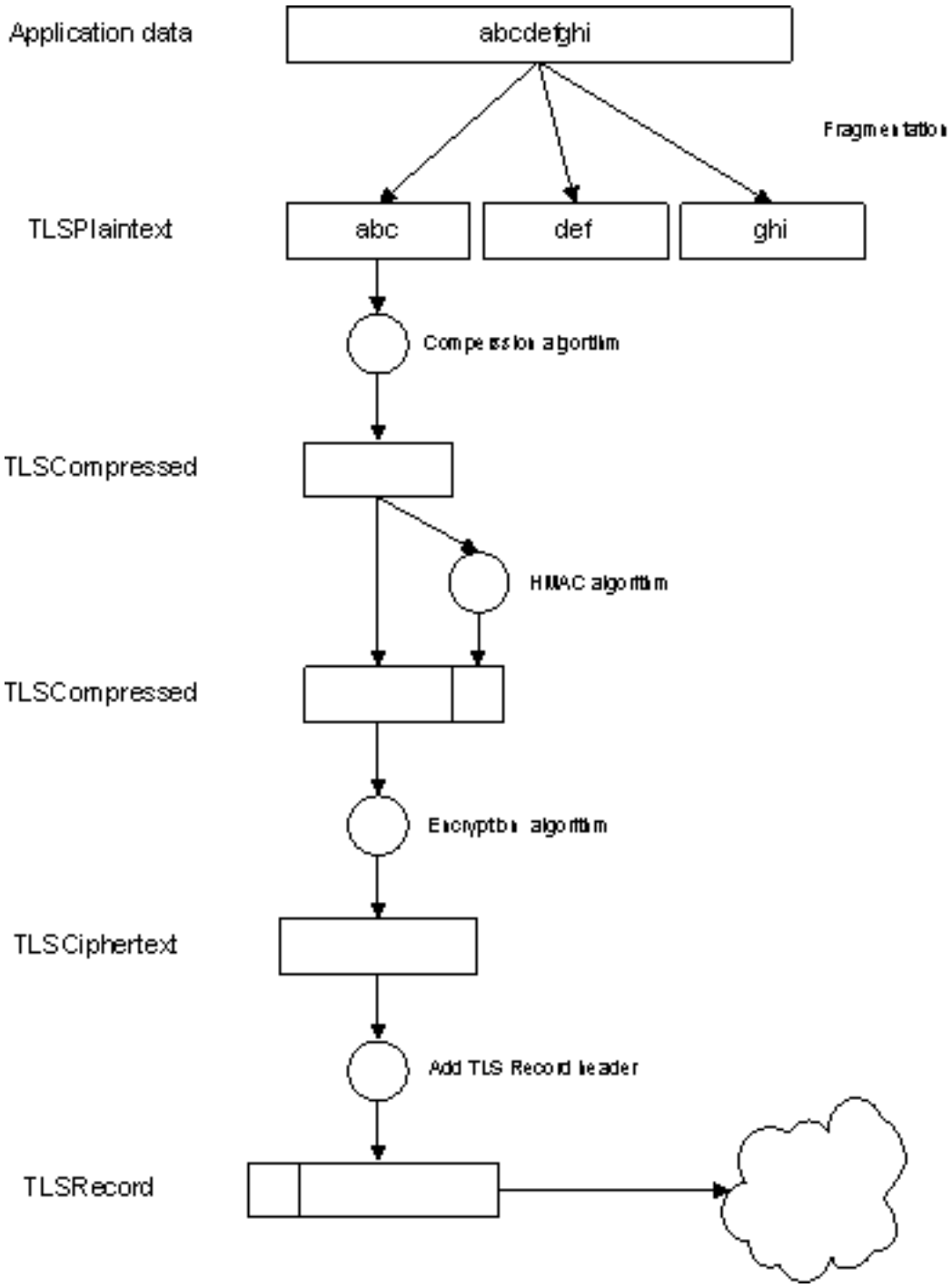


Figure 3.2: The Record Protocol

## 3.3 The Cipher Change Spec Protocol

This protocol consists of a single message of one byte, with the value '1', and is sent to inform the other party that all succeeding messages will be encrypted using the negotiated cipher suite and the calculated secret key. Both the client and the server must send the message before the session can be agreed as established. [7, 10]

## 3.4 The Alert Protocol

If there, at any point in the communication, a warning or an error occurs, an Alert protocol message is sent with the appropriate degree of seriousness and error code. The severity may be either fatal or warning. In case of fatal, the session is immediately terminated. The error codes are predefined and can be found in Appendix A.2. [7, 10]

## 3.5 The Handshake Protocol

The Handshake protocol is the most complex part of TLS, and is responsible for negotiating the secure attributes of a session. The messages sent during the handshake have a strict order they must follow, otherwise it will result in a fatal error. The table below provides a summary of the messages sent during a full handshake.

Table 3.1: TLS Handshake messages

| | | |
|---|---|---|
| 1. Client Hello | client ⇒ server | session id, nonce, supported cipher suites, supported compression methods |
| 2. Server Hello | server ⇒ client | session id, nonce, chosen cipher suite, chosen compression method |
| 3. Certificate* | server ⇒ client | certificate |
| 4. Key Exchange* | server ⇒ client | params, hash(params) |
| 5. Certificate Request* | server ⇒ client | certificate type, distinguished name |
| 6. Hello Done | server ⇒ client | (empty) |
| 7. Certificate* | client ⇒ server | certificate |
| 8. Key Exchange | client ⇒ server | encrypt(premaster secret) |
| 9. Certificate Verify* | client ⇒ server | hash(message 1-8) |
| 10. Change Cipher Spec** | client ⇒ server | (empty) |
| 11. Finished | client ⇒ server | prf(master secret, "client finished", message 1-9) |
| 12. Change Cipher Spec** | server ⇒ client | (empty) |
| 13. Finished | server ⇒ client | prf(master secret, "server finished", message 1-9) |

* This messages is optional
** This is not an actual handshake messages, but *Change Cipher Spec* protocol message. It is included because it has an important role in the handshake.

The handshake messages are according to the RFC5246[10], TLS Protocol version 1.2. The version has several improvements from previous versions, but the handshake steps

are identical. The most important improvements are added extensions ability, cipher suite independent pseudo-random function, hash signature algorithm is specified, and changes in the available and mandatory cipher suites. [10]

**1. Client Hello**
When a client connects to a server, the first message sent is the *ClientHello*. The client may also send this message at any time during a session if it wishes to exchange new security attributes.

The content of this message is as follows:

**Protocol version** The SSL/TLS protocol which the client wants to communicate by

**Random** Generated by the client with a secure random generator.

**Session ID** Used when the client wants to resume an old session. Empty if this is a new session or new security attributes should be generated.

**Cipher suite** A list of the supported cipher suites, sorted by the client's preference.

**Compression method** A list of the supported compression methods, sorted by the clients preference.

**2. Server Hello**
The server responds with a *ServerHello* message that contains the chosen parameters by the server. If no acceptable set of algorithms were found, the server will respond with a fatal error alert and the session will be terminated. In addition to the chosen protocol version, cipher suite and compression method, it contains the server random generated with a secure random generator. The ClientHello.random and ServerHello.random is completely independent of each other and will be used when generating the master secret. This message also contains the session identifier. If the ClientHello.session_id was not empty, the server will respond with the same session ID if it found a valid session with this ID. In this case, the handshake will proceed directly to the finished messages.

**3. Server Certificate** (Optional)
The server sends its digital certificate, generally a X.509v3 certificate. The certificate must have a key that matches the key exchange algorithm in the cipher suite. This message may contain more than one certificate; in that case the certificate must be listed as a certificate chain. If it is a chain, the first certificate must be the server's, and each succeeding must certify the preceding certificate. The chain is mostly used to authenticate the server by having a root certificate as the last in the chain. See more in Section 2.2.6. This message is optional, and sent only if the agreed key exchange method is not anonymous.

**4. Server Key Exchange** (Optional)
This message is sent only if the certificate does not contain enough information for the client to exchange the premaster secret. The server key exchange message must be sent when the key exchange methods is; DHE_DSS, DHE_RSA, or DH_anon.

**5. Certificate Request** (Optional)
The server can optionally request a certificate from the client. If sent, this message contains the certificate type and the distinguished name of the certificate authorities the server accepts.

**6. Server Hello Done**
The *ServerHelloDone* message indicates that the server is finished with its hello messages. At this point, the client should verify the server certificate, and whether it accepts the parameters from the server hello messages.

**7. Client Certificate** (Optional)
If the server sent the *CertificateRequest* message, the client must send this message in return. If the client does not have a valid certificate, it sends an empty *CertificateRequest* message; otherwise, it sends the certificate at the same format as described in the *ServerCertificate* message.

**8. Client Key Exchange**
Depending on the key exchange algorithm, and whether the client certificate message has been sent, this message contains either the premaster secret encrypted with RSA, or the client's Diffie-Hellman public-key parameters used to compute premaster secret, or empty if the client certificate message is sent and an appropriate key exchange method is used. In the first scenario, where the key exchange method is RSA, the client generates a premaster secret, which it encrypts with the public key from the server's certificate. In addition to the premaster secret or the public-key, this message contains the protocol version specified in the client hello message to detect roll-back attacks. The server must check that these two versions are identical before proceeding.

**9. Certificate Verify** (Optional)
If the client sent a certificate to the server, and the certificate has signing capability, this message contains a signature and is sent to verify the client certificate. The signature is a hash of the concatenation of all previous handshake messages, which are signed with the client's private key.

It is not necessary for the server to verify its certificate like this. If the server does not have the private key that belongs to the certificate, it cannot decrypt the premaster secret and will therefore be unable to generate the master key.

The master secret can now be computed with the PRF, explained in Section 3.6

**10. Client Change Cipher Spec**
The client sends this message to indicate that any succeeding message from the client will be encrypted with the decided cipher suite and the computed key. This is actually not a handshake message, but a Change Cipher Spec protocol message, consisting of a single byte '1'.

**11. Client Finished**
The finished message is sent by the client and used to verify that the key exchange and authentication process were successful, and that no previous message has been tampered with. This is the first message from the client that is encrypted with the negotiated parameters, and contains a PRF hash that the receiving part must verify. The secret sent to PRF is the master secret, the label is "client finished" and the seed is the SHA-256 value of a concatenation of all the previously exchanged handshake messages. The hash function, SHA-256, is the default hash function, but is cipher suite dependent. This means that the cipher suite may specify another, but stronger, hash function.

**12. Server Change Cipher Spec**
The server sends this message to indicate that any succeeding message from the server will be encrypted with the decided cipher suite and the computed key. This is actually not a handshake message, but a Change Cipher Spec protocol message, consisting of a single byte '1'.

**13. Server Finished**
The finished message is sent by the server and used to verify the key exchange and authentication process were successful, and that no previous message has been tampered with. This is the first message from the server that is encrypted with the negotiated parameters, and contains a PRF hash, explained in Section 3.6, that the receiving part must verify. The secret sent to PRF is the master secret, label is "server finished" and the seed is the SHA-256 value of a concatenation of all the previously exchanged handshake messages.

**Application Data**
At this point, the handshake is finished and application data can be sent securely between the parties.

## 3.6 PRF Computation

TLS uses a pseudorandom function (PRF) to calculate keys and to verify the handshake messages. The PRF takes a seed, a label, and a secret, and produces an output of arbitrary length. In TLS version 1.2, the PRF is cipher suite dependent. There is, however, only one PRF function specified, and it is stated that every new cipher suite MUST specify a PRF where the hash function is SHA-256 or stronger. The general form is:

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

The hash is SHA-256 or stronger, and P_hash is defined as:

P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) + HMAC_hash(secret, A(2) + seed) + ...

Where A(0) = seed, and A(i) = HMAC_hash(secret, A(i-1))

The function iterates as many times as necessary to produce the required length of output.

The key calculation is described below:

master_secret = PRF(premaster_secret, "master secret", ClientHello.random + ServerHello.random)
key_block = PRF(master_secret, "key expansion", ClientHello.random + ServerHello.random)

The key_block is partitioned into the following keys, according to their specified length;

client_write_MAC_secret
server_write_MAC_secret
client_write_key

server_write_key
client_write_IV
server_write_IV

Chapter 4

# The Implementation

*"The value of an idea lies in the using of it."*
- Thomas Alva Edison

The application is named EduTLS, and is a chat application where two hosts, hereafter called peers, can send text messages to each other. Before the peers can exchange messages, one of them must connect to the other. The application acts both as a server and a client; when the user initiates an outgoing connection to another peer, it has the role as a client. At the remote peer, that application has the role as the server. These roles, however, are just during the connection setup; when it has been established, the roles have no practical meaning anymore. The application is developed in Java. A brief description of Java can be found in Section 2.4.1. The complete source code can be found at https://github.com/evinje/EduTLS.

When using the EduTLS application to study the protocol, it is not required to know the Java programming language. However, the benefits are much greater when looking at the source code in addition to the graphical user interface. The intention is to study both the source code and the application. By doing that, one can follow the flow between components, do modifications, add functionality and so on. Another feature is the ability to add more cryptographic primitives and cipher suites. These could be self-designed or other implementations of the same primitives, and test its effectiveness with the built-in performance testing tool, or used to perform cryptanalysis.

Unless otherwise specified, when referring to the TLS protocol in this chapter, it refers to the simplified TLS protocol, which is designed and implemented in this project.

## 4.1 Requirements

This section states the requirements for the application, in prioritized order.

Table 4.1: Requirements

1. **Simplify** the application shall make it easier to understand TLS
2. **Java** the application shall be implemented in Java
3. **Architecture** the architecture shall be similar to TLS
4. **Handshake** the handshake shall have the same steps as TLS
5. **Record** the record protocol shall have the same responsibility as in TLS
6. **Resume** it shall be possible to resume an old session
7. **Time measure** the GUI shall show how time consuming the operations are
8. **Automated tests** the project shall include automated tests
9. **X.509v3** there shall be X509v3 similar certificate
10. **CA** there shall be a CA that can sign and verify certificates
11. **Miller-Rabin** there shall be the Miller-Rabin primality test to generate primes
12. **Rijndael** the Rijndael algorithm shall be implemented
13. **RSA** the RSA public-key encryption shall be implemented
14. **DH** the DH key exchange shall be implemented
15. **DES** the DES algorithm shall be implemented
16. **SHA-1** the SHA-1 hash function shall be implemented
17. **SHA-256** the SHA-256 hash function shall be implemented
18. **PRNG** there shall be at least one random number generator
19. **Compression** there shall be at least one compression method
20. **Performance test** there shall be a performance test tool

These requirements were made before starting on the implementation, and are more like goals than strict requirements. The goals have been taken into consideration when designing the system, when decisions and priorities had to be made, and is discussed in the evaluation conducted in Chapter 5.

## 4.2 Overall Approach

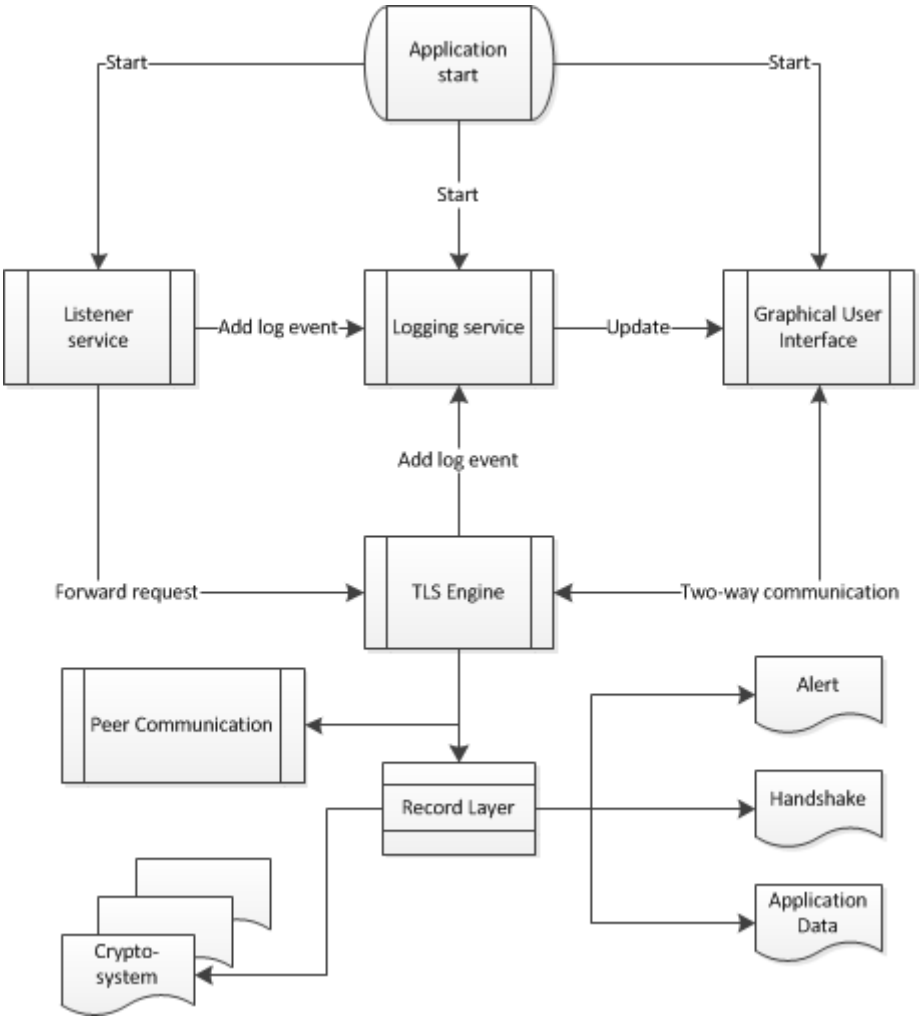


Figure 4.1: Flow chart

Figure 4.1 shows an overview of the flow between core components. At application startup, the listener service, logging service, and the graphical user interface is created. When the listener service receives an incoming request, the type of request is determined. This may be *CONNECTION_TYPE_TEST* or *CONNECTION_TYPE_TLS*. In case of the former, the listener service responds with a successful test message[1], and in case of the latter, the request is forwarded to the TLS Engine. The TLS Engine creates a log event for every operation it performs, and these log events are sent to the logging service. Whenever the logging service receives a log event, it notifies the graphical user interface. By having this separation, the graphical user interface has no direct coupling to the protocol and can easily be replaced with another user interface.

TLS Engine and the graphical user interface have two-way communication; when the user connects, or sends a message, to the remote peer, the graphical user interface module transmit the request to TLS Engine. At the remote peer, TLS Engine receives the re-

[1]It simply replies with a *CONNECTION_TYPE_TEST* message

quest from the listener service, and after processing the message, it is transmitted to the graphical user interface.

TLS Engine has two other dependencies in addition to the logging service and the graphical user interface. These are the peer communication system and the record layer. The record layer is used whenever an object is to be transmitted or an object is received. The record layer interpret the content type of the object, which can be Alert, Handshake or Application Data. Depending on the content type, the TLS Engine determines the further process of the object. The peer communication system is responsible for the transport of objects between two peers. This involves both sending objects and receiving objects over a network connection.

All of these components are explained in more detail throughout the rest of this chapter.

## 4.3 The Graphical User Interface

### 4.3.1 Conceptual View

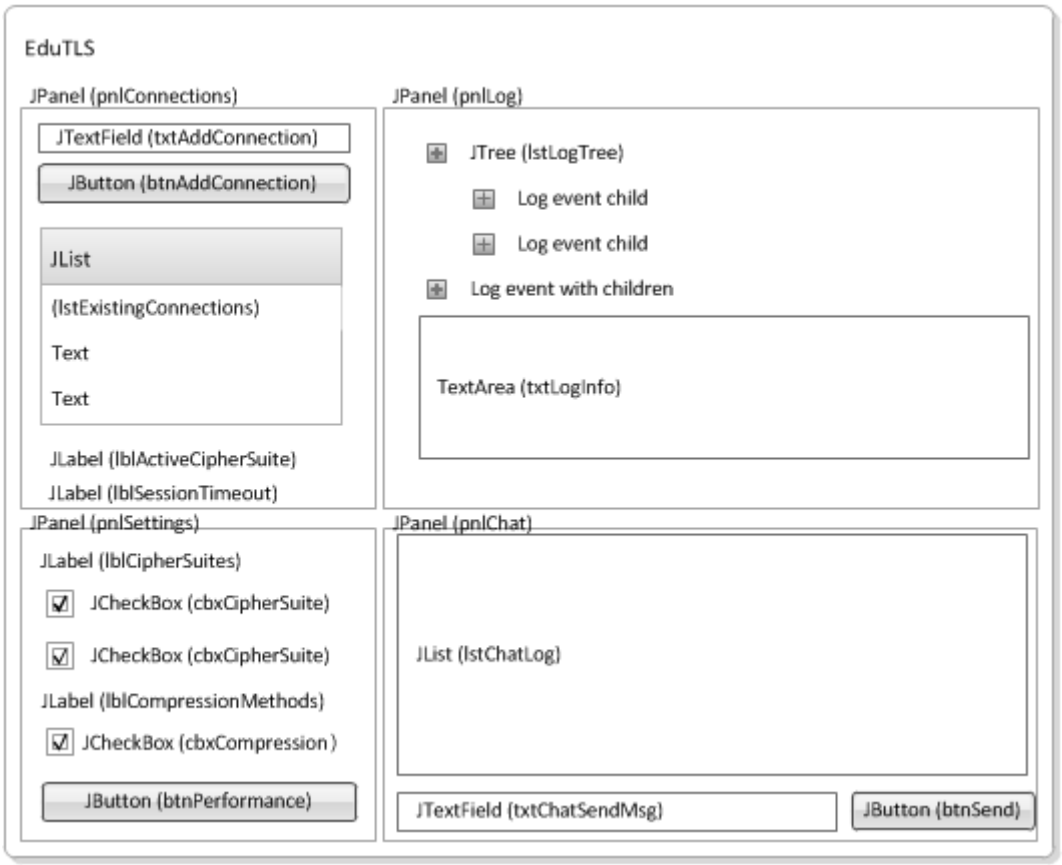


Figure 4.2: The GUI - Conceptual view

Figure 4.2 demonstrates a conceptual view of the graphical user interface. The conceptual view contains every component in the graphical user interface, with both its type and

name. The type refers to the Java Swing component name, and the self-determined name is given in parentheses. This information only has importance when studying the source code.

The graphical user interface is separated into four panels. The upper left panel, *pnlConnections*, contains information about the connections. The top of the panel gives the user the opportunity to add new connections. Adding a connection is accomplished by entering the hostname or IP address to the remote host into the *txtAddConnection* text field, and click the *btnAddConnection* button. Below is information on connections, both passive and active, listed in the *lstExistingConnections* JList. It is only possible to have one active connection at the time, and the active connection is the one that is selected in the list. At the bottom of the panel is information about the active cipher suite for the current connection, and countdown timer of when the active session expires. More information about sessions can be found in Section 4.4.5.

The upper right panel, *pnlLog*, contains log information. The JTree *lstLogTree* keeps the log organized hierarchically, and when clicking on one of the nodes in the tree, a detailed description of the log event is displayed in the text area *txtLogInfo* in addition to expanding the node if it has children. More information about the log is found in Section 4.4.3.1.

The bottom left panel, *pnlSettings*, lists all available cipher suites with a check box in front of the cipher suite name. The check box indicates whether the cipher suite is supported or not. When connecting to, or receiving connection from, a remote peer, the active cipher suite is the best available from both the peers. The active cipher suite is chosen during the handshake, explained in Section 3.5. Below the cipher suites are the available compression methods, which also is negotiated during the handshake. The settings panel also have a button, *btnPerformance*, which tests the performance of all the cryptographic primitives available. More information about the performance test is found below Figure 4.7.

The last panel, *pnlChat*, located at bottom right, contains a log history of the chat, a text field and a button. The log history text field, *lstChatLog*, lists all messages that is sent and received. When the user wants to send a message, the message is written in the text field *txtChatSendMsg*, followed by clicking the *btnSend* button.

### 4.3.2 Startup

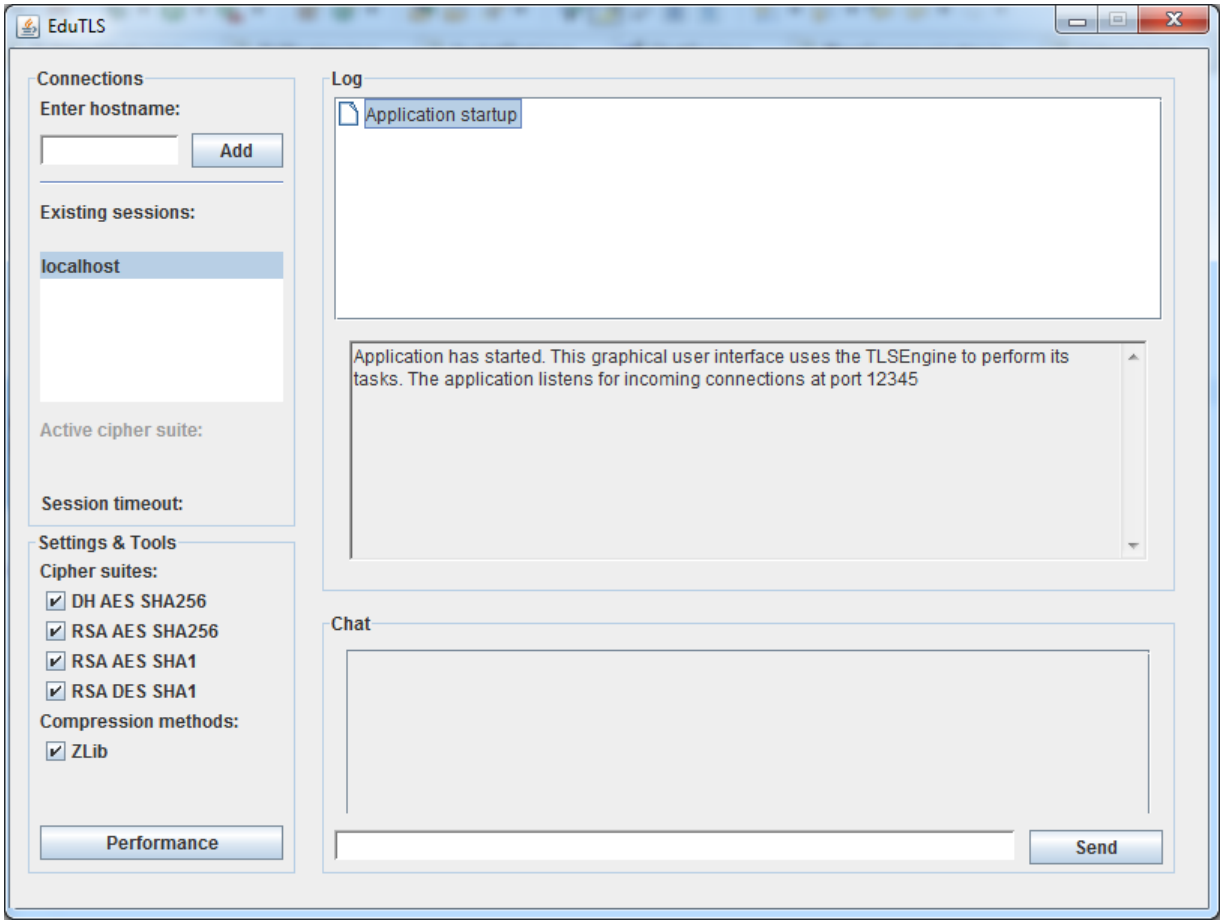


Figure 4.3: The GUI - Startup

Figure 4.3 shows the graphical user interface that is presented to the user when running the application. A log event, "Application startup" is created, giving the user information that the application waits for incoming connections at port 12345. All available cipher suites and compression methods are loaded and enabled, and displayed in the settings panel. Under existing sessions, *localhost* is added as a "dummy-connection". If no other host is known, *localhost* can be connected to as an ordinary connection. The *localhost* connection is the same computer as the EduTLS application runs at. This means, if the user connects to *localhost*, the application is both the client and the server at the same time. The log event viewer clearly shows this, because all transmitted packages are displayed twice, one as sent and one as received.

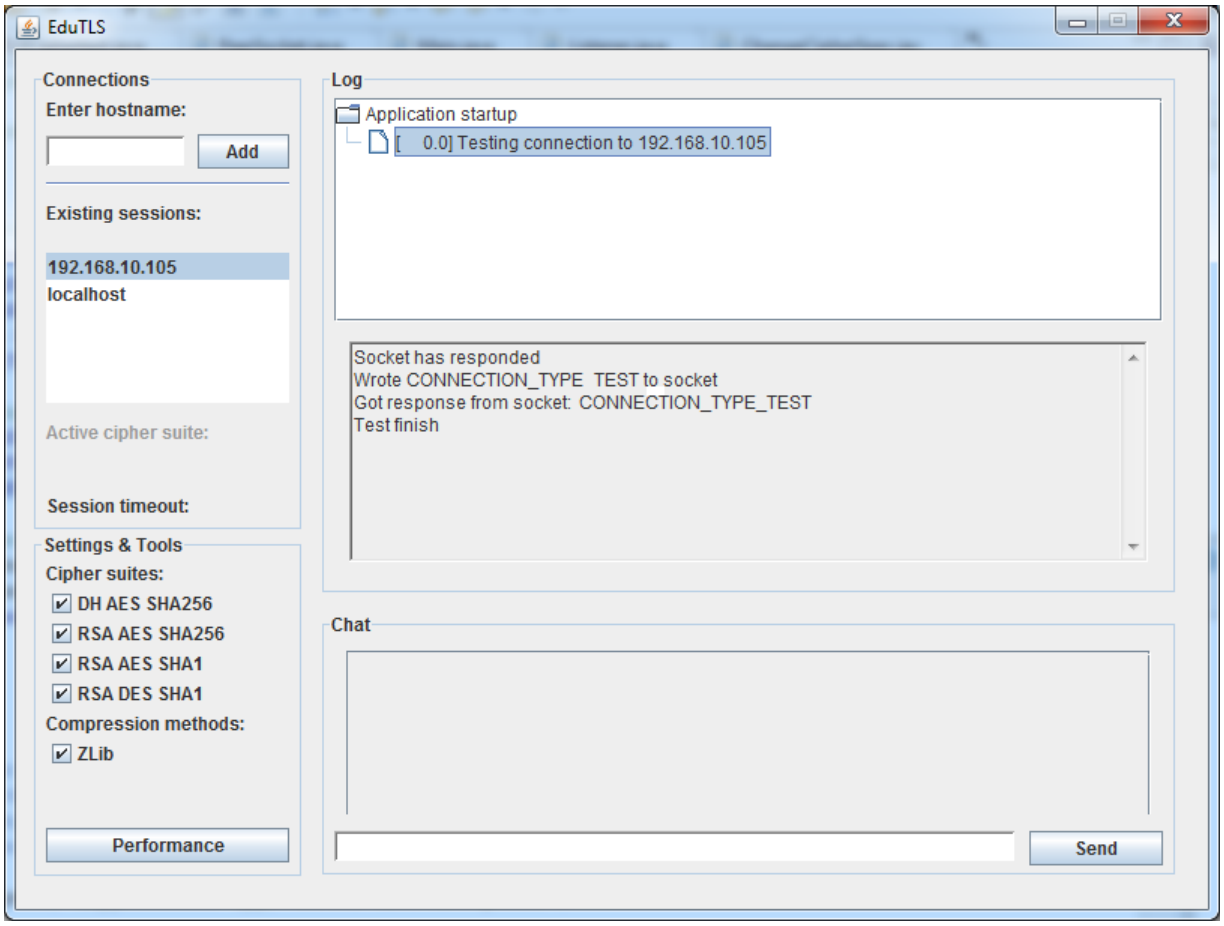


Figure 4.4: The GUI - Add new connection

When a new connection is added, the application creates a test connection to the remote peer, and if the test was successful, the host is added. More information about the test connection is given in Section 4.4.2.1.

### 4.3.3 Connect

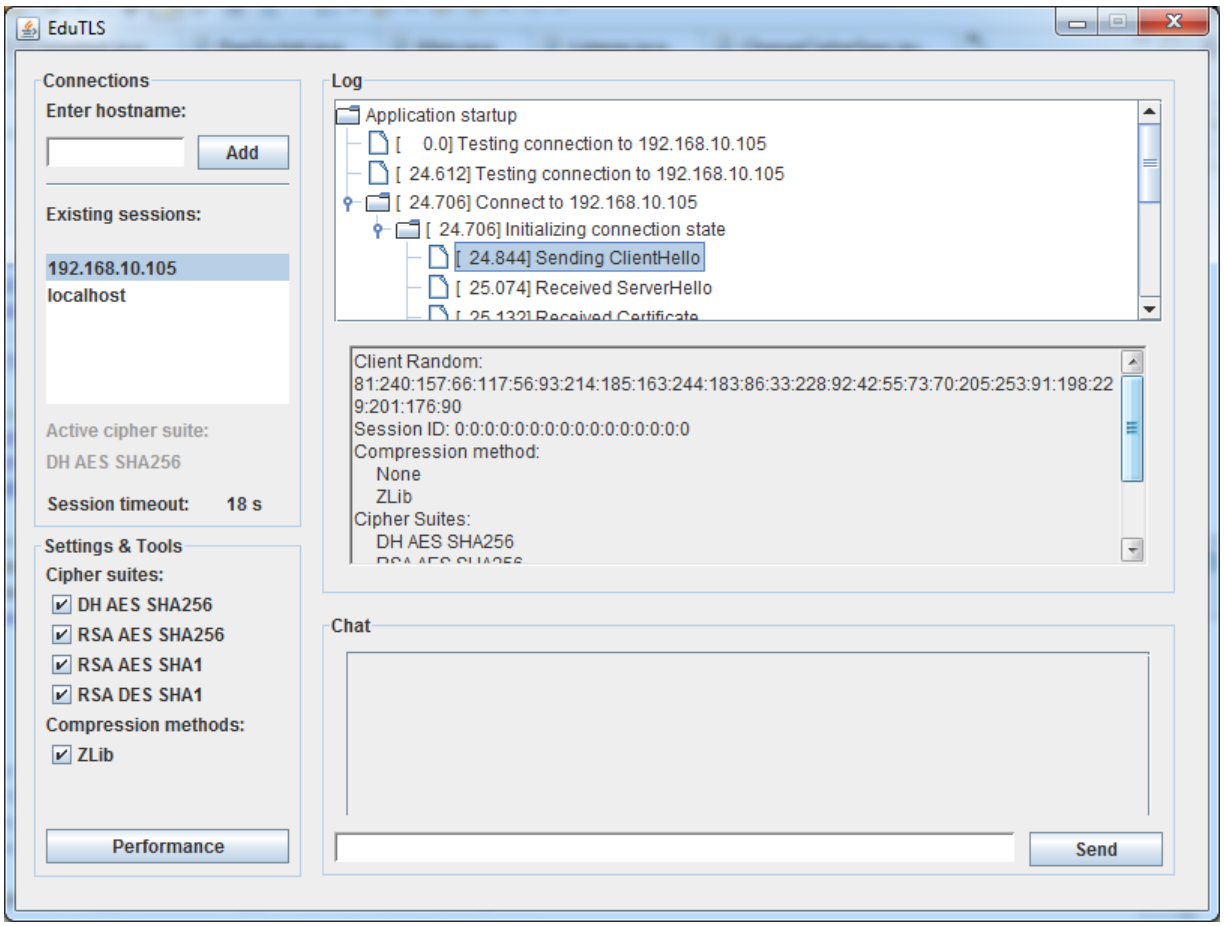


Figure 4.5: The GUI - Connect

When the user selects a host in the list of existing sessions, the system creates a connection to that host. The graphical user interface calls the *TLSEngine.connect()* method, which initializes a handshake with the remote peer.

Following are several listings, demonstrating a real-life handshake from the client. Below every listing there is given an explanation of the content.

```
1  Client Random:
2  242:209:203:4:56:104:176:136:215:3:222:60:69:138:38:162:131:
3  60:37:42:227:30:117:106:95:244:15:108
4  Session ID: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0
5  Compression method:
6          None
7          ZLib
8  Cipher Suites:
9          DH AES SHA256
10         RSA AES SHA256
11         RSA AES SHA1
12         RSA DES SHA1
```

Listing 4.1: Sending ClientHello

The client hello consists of a client random, session id, compression method and cipher suites. The client random is generated with the *randBytes* method of the BlumBlumShub class, explained in Section 4.4.4.5. In this example, the session id is empty, indicating that the client have no previous connections to the server. EduTLS supports *None* and *ZLib* as compression algorithm. There are currently four cipher suites supported by EduTLS; *DH AES SHA256*, *RSA AES SHA256*, *RSA AES SHA1*, and *RSA DES SHA1*. These are listed in prioritized order.

```
Server Random:
127:185:99:100:3:123:144:187:130:242:237:74:230:44:115:109:
124:30:44:131:194:225:199:104:227:100:162:1
Session ID:
166:210:13:69:53:158:198:74:244:209:186:187:134:189:218:184
Compression method: ZLib
Chosen Cipher Suite: DH AES SHA256
```

Listing 4.2: Receive ServerHello

The server responds with server hello, which is very similar to the client hello. The server random is generated by the same method as the client random. The client provided no session id, making the server generate a new session id, and includes this value in the response. The chosen cipher suite is the best cipher suite that both the client and the server supports, *DH AES SHA256* in this example, with the ZLib compression method. The compression method is not a part of the cipher suite, as specified in Chapter 3.

```
Certificate:
Data:
Version:1
Serial Number:8
Issuer:CN=EduTLSv2, O=NTNU, L=Trondheim, S=SorTrondelag, C=NO
Validity:
  Not Before:Tue May 10 13:49:18 CEST 2011
  Not After:Fri June 10 13:49:18 CEST 2011
Subject:CN=192.168.10.104
Public Key Algorithm:DiffieHellman
  Public Key (512 bit):
  Modulus (512 bit):
    447706644435888682499618224329283239257656104107819594369
    964488581742552394273270889686576752518613837954250209486 85
    638887108893845530525833596716734220 45
  Exponent:
    551801708337915270629554126389150892239274964261236420344 1
    090463562024824763672462873355102798956634559469333235840 9
    520674604455159394146085930301702551 55
  Signature Algorithm:sha1WithRSAEncryption
    215979267288372513282690761578529048136346602707191769346 8
    268712938626610548793026186949802790337979852254373564212 4
    329618794641099422642641591429344420412674569392336726096 4
    574501416778664489460956102244275084940081679259183070774 4
    850332565261877844921580471218772011593117462071033704413 6
    298431961094336144074572866489266451138893902086238290005 7
    827890347613436809387971512677955621593315893194177438276
    070862317680256652721261959828395304032500030282114474399 5
    977004800365684465710658512064085427464747756247862303784 6
```

```
30      170761114561020863305253051197017481660939377692941538852 6
31      217438083693944864636099141123957067
```

Listing 4.3: Receive Certificate

The server certificate is closely related to the X509 certificate[19], described in Section 2.2.6. The certificate is generated at every startup of the application. In a functional environment however, the certificate is generated only once, saved on a permanent storage location, and loaded at startup of the application. The validity period is one month, which should be plenty for testing purposes. As the certificate in this demonstration states, the public key algorithm is Diffie-Hellman, while the signature algorithm is SHA-1 with RSA encryption. The public key algorithm must correspond to the one in the cipher suite negotiated in the hello messages. Like in the X.509v3 certificate, these algorithm does not need to be the same, since the signature comes from the certificate authority. Note that the common name (CN) is the IP address for the remote peer, which is not sufficient from a security perspective, since these can easily be tampered.

```
1 55:48:49:51:56:57:50:54:56:48:57:56:56:52:57:56:50:55:57:
2 49:56:56:53:53:52:48:50:51:52:52:57:54:53:55:49:55:48:52:
3 53:49:48:49:49:48:49:57:53:55:49:56:50:50:56:52:51:53:50:
4 53:51:56:56:49:57:55:53:53:53:55:57:54:53:56:54:56:51:52:
5 56:57:52:53:48:57:53:51:50:51:49:52:49:55:55:50:52:57:51:
6 55:54:54:54:48:48:56:52:48:54:50:52:54:51:54:55:54:53:53:
7 52:52:54:57:49:51:56:55:49:49:57:54:52:49:56:48:53:49:52:
8 51:56:52:55:49:55:55:55:55:50:48:48:57:52:55:50:54:55:52:
9 49:55
```

Listing 4.4: Receive ServerKeyExchange

Since the current cipher suite of this example uses the Diffie-Hellman key exchange algorithm, the server must send the server key exchange message in addition to the certificate. This message contains the Diffie-Hellman base generator.

```
1 <empty message>
```

Listing 4.5: Receive ServerHelloDone

After the server key exchange message, the server sends server hello done, indicating that the server has finished its messages and waits for response from the client. The server hello done message is an empty message.

```
1 Algorithm: DiffieHellman
2 Public Key:
3 29373395585607785758556408459945677838692064268190069981 1
4 09662960161124110804051159538065257423386585119857318445 6
5 03860155419975726810137713487045393821 5
6 Modulus:
7 74686000784664261147964638241255264122031066126097902924 6
8 61068049572941778667761674991030111022362018125495276339 4
9 06595310117328480770789113422946212476 79
```

Listing 4.6: Send ClientKeyExchange

The second message the client sends is the client key exchange message. This message contains the public key and modulus of the clients Diffie-Hellman key pair. When the server receives the client key exchange message, both parties have enough information to generate the master secret. If the public-key algorithm in the certificate was RSA, this message would be the premaster secret, encrypted with the public-key from the certificate.

```
Total size of key block: 128
The seed (server random and client random concatenation):
127:185:99:100:3:123:144:187:130:242:237:74:230:44:115:109:
124:30:44:131:194:225:199:104:227:100:162:1:242:209:203:4:
56:104:176:136:215:3:222:60:69:138:38:162:131:60:37:42:227:
30:117:106:95:244:15:108
Pre-master secret (64 bytes):
5:155:189:222:0:7:121:29:51:222:16:178:184:71:110:131:131:
152:172:95:149:103:123:61:6:55:14:210:91:137:136:255:61:89:
114:54:213:59:251:158:50:228:158:226:152:86:169:33:187:176:
67:7:102:1:128:157:216:44:203:254:168:197:232:39
Master secret (48 bytes):
191:212:125:34:108:188:95:128:119:207:162:100:157:102:157:
16:40:189:8:5:1:9:0:7:0:0:2:21:0:4:167:0:116:252:4:201:21:
66:154:226:65:124:107:26:230:248:145:2
Client write mac key (32 bytes):
46:252:246:109:114:147:23:2:134:231:125:103:15:133:237:71:
121:107:95:174:0:9:51:124:1:200:67:123:241:11:194:54
Server write mac key (32 bytes):
187:246:255:138:210:122:28:20:180:54:180:131:235:187:50:224:
159:152:174:31:255:126:166:14:15:174:167:54:151:96:211:1
Client write encryption key (16 bytes):
168:35:134:204:105:201:128:82:101:134:110:182:228:84:111:180
Server write encryption key (16 bytes):
112:203:38:25:118:215:215:193:14:114:132:117:253:189:197:9
Cliejnt write IV (16 bytes):
128:165:161:149:246:78:213:222:47:1:120:194:83:15:162:159
Server write IV (16 bytes):
17:99:240:212:145:43:175:9:44:21:230:140:109:213:157:71
```

Listing 4.7: Generating Key Block

The key block generation is not a part of the exchanged messages, but is included to demonstrate at what point the session keys are generated. When using the *DH AES SHA256* cipher suite, the size of the key block is 128 bytes. After generation, the key block is partitioned into 2 x 32 bytes for client and server write mac keys, 2 x 16 bytes for client and server write encryption keys, and 2 x 16 bytes for client and server write IV. The IV is generated because of the mode of operation, see Section 2.2.1.

```
1
```

Listing 4.8: Send ChangeCipherSpec

The client follows the client key exchange message with the change cipher spec message. This is not a handshake message, but is a vital part of the handshake because it indicates that every succeeding message is encrypted with the current state algorithms and keys.

```
PRF(master_secret, "client finished", Hash(handshake_messages))

Where master_secret =
191:212:125:34:108:188:95:128:119:207:162:100:157:102:157:
16:40:189:8:5:1:9:0:7:0:0:2:21:0:4:167:0:116:252:4:201:21:
66:154:226:65:124:107:26:230:248:145:2
and handshake_messages =
      ClientHello
      ServerHello
      ServerCertificate
      ServerKeyExchange
      ServerHelloDone
      ClientKeyExchange
```

Listing 4.9: Send Finished

The last message from the client in the handshake is the finished message. The finished message is calculated from every previous handshake message, and reveals if any of the handshake messages have been tampered with. The finished message is generated with the PRF method, described in Section 3.6. The seed to the PRF method is a concatenation of every handshake message, except any hello request message. It is important to note that the change cipher spec message is not a handshake message; hence it is not included in the seed.

```
1
```

Listing 4.10: Receive ChangeCipherSpec

```
PRF(master_secret, "server finished", Hash(handshake_messages))

Where master_secret =
191:212:125:34:108:188:95:128:119:207:162:100:157:102:157:
16:40:189:8:5:1:9:0:7:0:0:2:21:0:4:167:0:116:252:4:201:21:
66:154:226:65:124:107:26:230:248:145:2
and handshake_messages =
      ClientHello
      ServerHello
      ServerCertificate
      ServerKeyExchange
      ServerHelloDone
      ClientKeyExchange
```

Listing 4.11: Finished

After the final message from the client, the server responds with the change cipher spec message, and the finished message. The only difference between the finished messages is the label, where the client provides ”client finished”, and the server provides ”server finished” to the PRF method.

At this point, the handshake is finished and the peers are connected.

### 4.3.4 Send Message

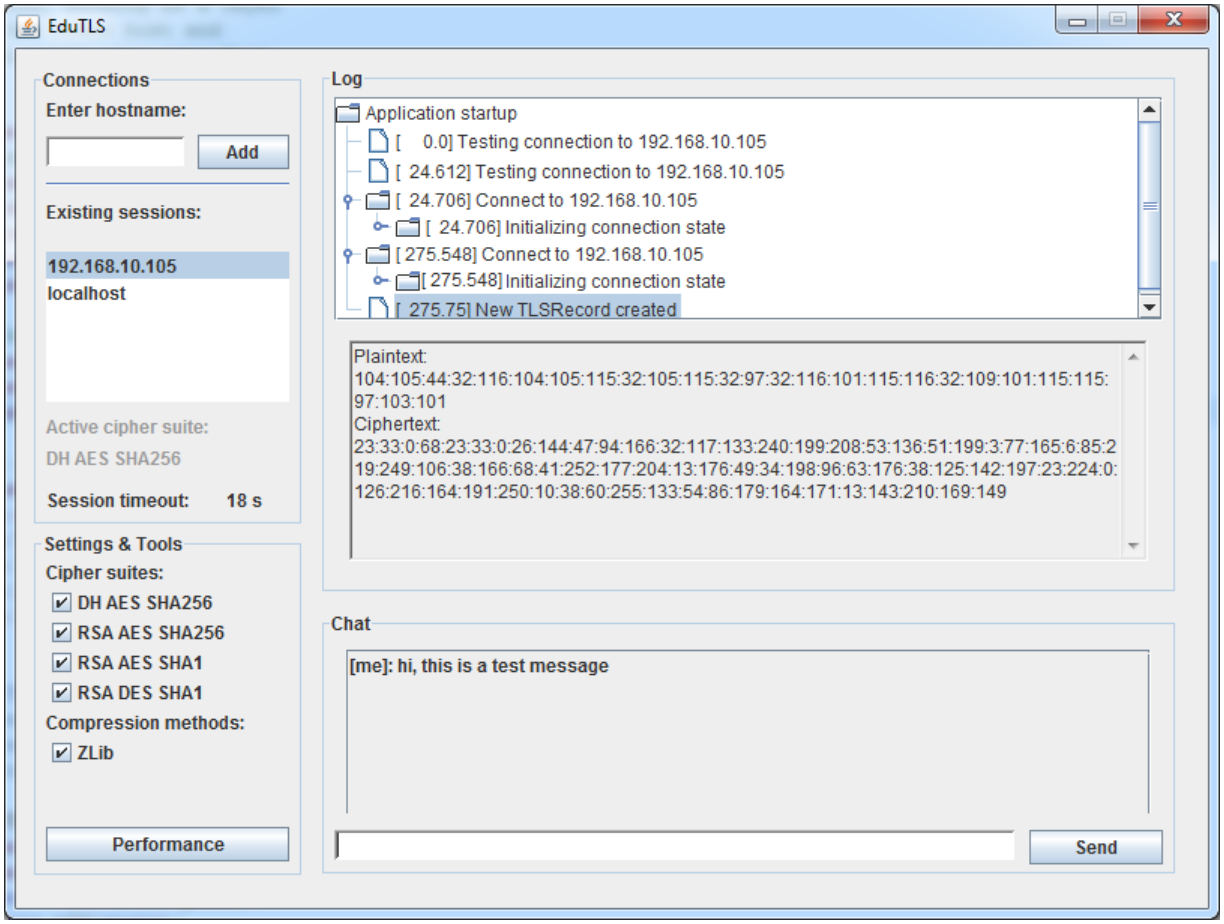


Figure 4.6: The GUI - Send message

After the connection has been established, the user may enter a message and click the send button. In the log, both the plaintext and ciphertext of the message is presented. Note the active cipher suite is displayed, and a countdown timeout to the session is closed. When the timeout has reached 0, a new handshake, with session resume, must be initiated before the peers can exchange more messages. The ciphertext in the figure is actually the TLSRecord object that is sent. The first four bytes, *23:33:0:68*, is the Record header, where 23 specified application data, 33 is the version number, and 0:68 is the total size in bytes. The next four bytes are the TLSCiphertext header, where the two first are identical to the record header in this example. The next two, 0:26, specifies that the ciphertext is 26 bytes. The remainder is padding to obtain correct block size of the encryption algorithm.

### 4.3.5 Performance Test

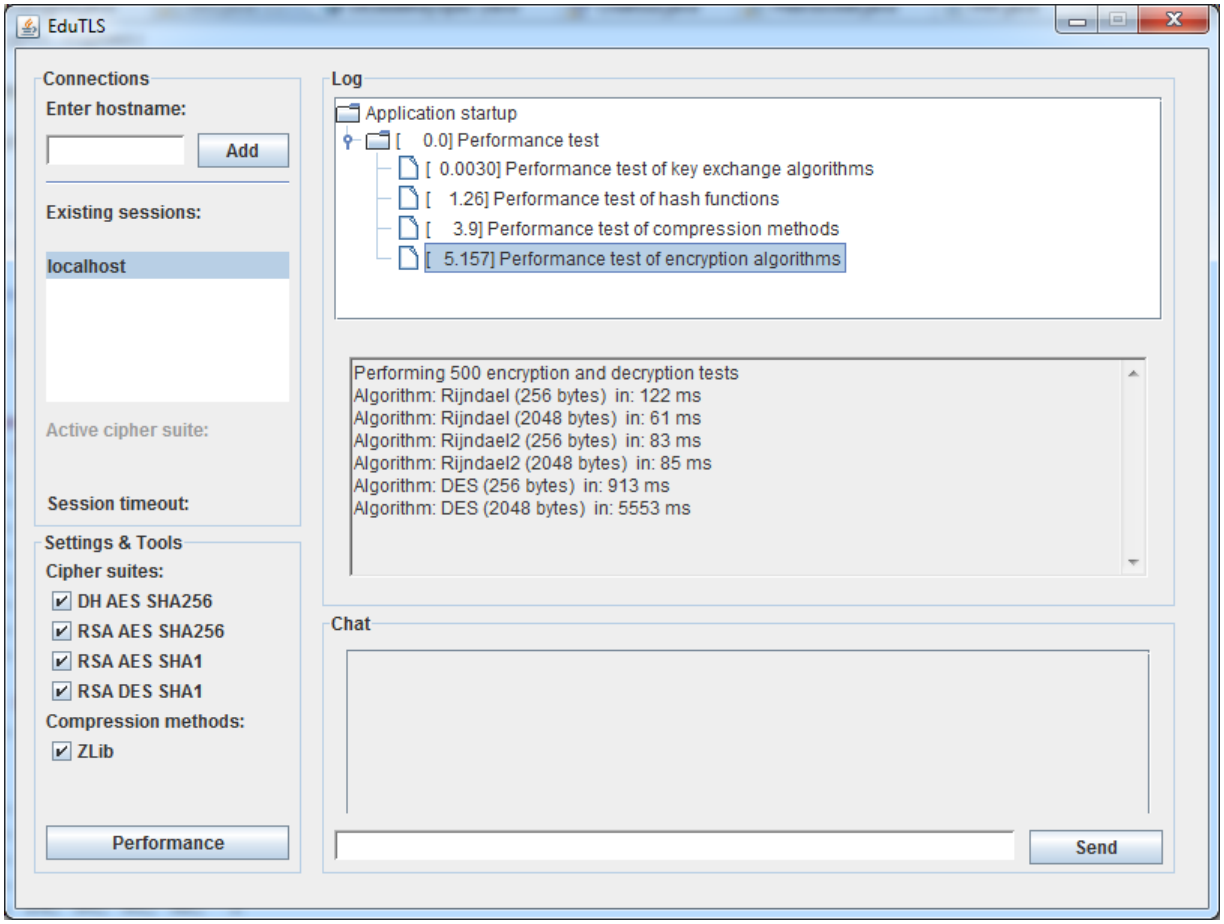


Figure 4.7: The GUI - Performance test

The performance test goes through every cryptographic primitive, and tests their performance. The log provided from a performance test is demonstrated in the listing below.

```
1 Key exchange algorithms, 500 tests
2 Generated 512 bits DiffieHellman keys in: 3 ms
3 Algorithm: DiffieHellman in 582 ms
4 Generated 512 bits RSA keys in: 528 ms
5 Algorithm: RSA in 56 ms
6
7 Hash functions, 5000 tests
8 Algorithm: SHA-1 (16 bytes) in: 33 ms
9 Algorithm: SHA-1 (512 bytes) in: 48 ms
10 Algorithm: SHA-1 (16384 bytes) in: 1230 ms
11 Algorithm: SHA-256 (16 bytes) in: 62 ms
12 Algorithm: SHA-256 (512 bytes) in: 81 ms
13 Algorithm: SHA-256 (16384 bytes) in: 1363 ms
14
15 Performing 1000 compression tests (compress and decompress)
16 Algorithm: ZLib (16 bytes, 31.25% compression ratio) in: 258 ms
17 Algorithm: ZLib (512 bytes, 97.3% compression ratio) in: 276 ms
18 Algorithm: ZLib (16384 bytes, 99.8% compression ratio) in: 746 ms
19
20 Conventional encryption, 500 tests (both encryption and decryption)
```

```
21 Algorithm: Rijndael (256 bytes)  in: 122 ms
22 Algorithm: Rijndael (2048 bytes)  in: 61 ms
23 Algorithm: Rijndael2 (256 bytes)  in: 83 ms
24 Algorithm: Rijndael2 (2048 bytes)  in: 85 ms
25 Algorithm: DES (256 bytes)  in: 913 ms
26 Algorithm: DES (2048 bytes)  in: 5553 ms
```

Listing 4.12: Performance Test

The first step in the test is the key exchange algorithms. There are currently two algorithms specified; DiffieHellman and RSA. The key exchange algorithms are tested by doing 500 encryptions and one key pair generation. The Diffie-Hellman key generation is performed in 3 milliseconds, but this is with pre-generated $P$ and $G$ values. Appendix B.4 demonstrates how the $P$ and $G$ values have been generated. The RSA key generation takes 528 ms, but this involves every step in the key-pair generation, including prime generation with the miller rabin test. Appendix B.5 demonstrates both miller-rabin primality test and the RSA key generation.

The next test is the hash functions, where 5000 tests are performed on three different sizes. Since the number of tests is quite large, there will be a little overhead because every hash input is different. From the test in the demonstration, the SHA-1 algorithm has much higher performance than SHA-256 when processing short texts, but at larger texts, they have more similar execution time. The crypto++ benchmarks[45] shows similar results, where the SHA-1 has about 37% higher performance than SHA-256.

The only compression method available is ZLib. The TLSv1.2 specification states no compression method[2], but it has been decided to implement a compression method in EduTLS. The compression method test runs for 1000 cycles, and compresses three different text sizes. The performance test shows the compression ratio in addition to the time spent. Since the texts are not randomly generated, the compression ratio is extremely high. The intention of the performance test is not test the efficiency of the compression ratio, but it could certainly be an improvement goal, and is mentioned in Section 5.3.

The final step is to test the conventional encryption algorithms. There are three conventional encryption algorithms; Rijndael, Rijndael2 and DES. The first two classes are the Rijndael/AES algorithm, but different implementations of it, and have been included to examine the difference in execution time between implementations. The first Rijndael algorithm is a part of the bouncy castle project[20], whereas the second (Rijandel2) is taken from the GNU crypto project. The DES algorithm is developed by Dr. Herong Yang, and according to the test demonstrated, it has very bad performance.

[2]Besides CompressionMethod.null

## 4.4 Classes and Packages

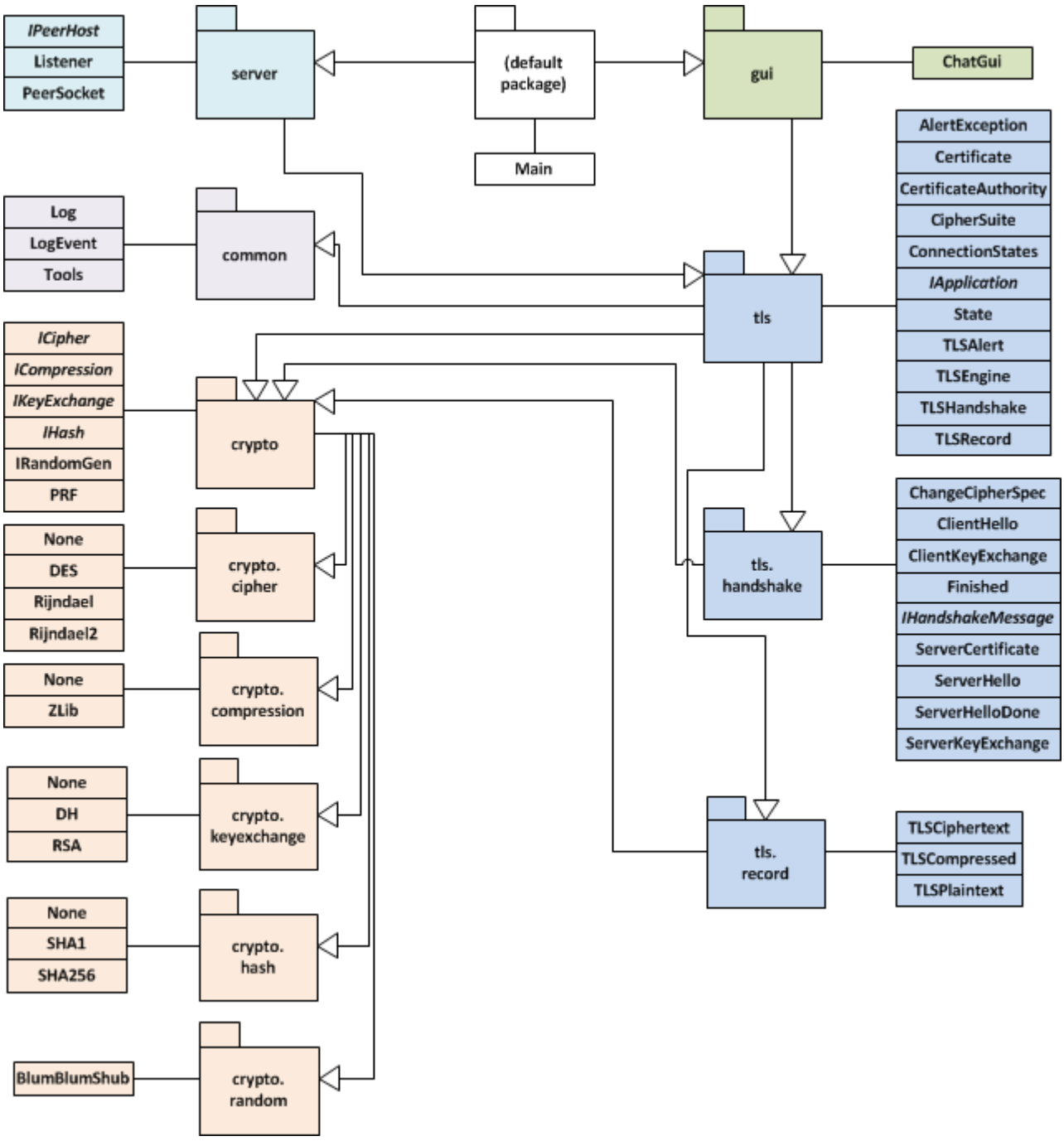


Figure 4.8: Packages and classes

The large squares in Figure 4.4 are packages, while the smaller and rectangular are classes and interfaces. The names in italic are interfaces. As the figure demonstrates, the TLS engine communicates with the interfaces located in the crypto package, and is completely unaware the actual algorithm it uses.

Table 4.2: Source code size summary

| Package name | Number of classes | Size |
|---|---|---|
| (none) | 1 | 32 lines (20 without comments) |
| common | 3 | 335 lines (193 without comments) |
| crypto | 6 | 71 lines (49 without comments) |
| crypto.cipher | 4 | 1848 lines (1445 without comments) |
| crypto.compression | 2 | 113 lines (88 without comments) |
| crypto.hash | 3 | 423 lines (283 without comments) |
| crypto.keyexchange | 3 | 339 lines (231 without comments) |
| random | 1 | 181 lines (138 without comments) |
| gui | 1 | 671 lines (546 without comments) |
| server | 3 | 429 lines (332 without comments) |
| tls | 10 | 1456 lines (1063 without comments) |
| tls.handshake | 9 | 1145 lines (925 without comments) |
| tls.record | 3 | 513 lines (392 without comments) |
| Total | 49 | 7000 lines (5261 without comments) |

The table above lists all packages and their corresponding size in means of classes and number of lines of code. The next sections explains all these packages and their classes, with a table at the end of each section with the size of the source code. These tables are included to demonstrate the complexity of each class in the package. The size of a class is not necessarily a measure of its complexity, but in this project it reflects where the operations are located.

### 4.4.1 The GUI Package

The GUI package consist of only one class; ChatGui. This is the source code for the graphical user interface of the application. A description of the graphical user interface is given in Section 4.3. It is worth mentioning that it has been made compatible with GUI design tools, for instance WindowBuilder[3]. Using GUI tools is a good option for those who are not familiar with the Java Swing toolkit, but want to improve or experiment with the graphical user interface. GUI design tools are WYSIWYG[4] editors, letting the user drag-and-drop components and visualize the result live, instead of writing code that generates the graphical user interface.

Table 4.3: The GUI package source code

| Class name | Size |
|---|---|
| ChatGui | 671 lines of codes (546 without comments) |

[3] http://code.google.com/javadevtools/wbpro/index.html
[4] What You See Is What You Get

### 4.4.2 The Server Package

The server package contains functionality that deals with the incoming connections service and communication.

#### 4.4.2.1 Listener

The Listener class creates a server socket which waits for incoming connections. When another peer connects, the listener service examines the type of request. If the request is *CONNECTION_TYPE_TEST*, the listener service responds with a *CONNECTION_TYPE_TEST* message, meaning the test was successful. In the other type of request, *CONNECTION_TYPE_TLS*, the listener service creates a TLSRecord of the request, and forwards the record object to the TLSEngine.

#### 4.4.2.2 IPeerCommunicator

IPeerCommunicator is an interface for the communication that EduTLS requires.

```
1 public interface IPeerCommunicator {
2    public String getPeerId();
3    public TLSRecord read(State state);
4    public void write(TLSRecord record);
5    public void close();
6    public boolean reconnect();
7    public boolean isClient();
8    public boolean isConnected();
9 }
```

Listing 4.13: IPeerCommunicator Interface

The listing above is the IPeerCommunicator interface. It defines sending and receiving of TLSRecord objects, and other basic communication features, for instance reconnect and close. This means that one can create several different communication methods for the application. The EduTLS only have implemented the standard network communication method, called socket communication, but could be implemented otherwise for testing purposes. It is also possible to make the communication go through a proxy to examine the communication. The automated tests has its own implementation of the IPeerCommunicator, making the tests more efficient and isolated, this is explained more in Section 4.6.

#### 4.4.2.3 PeerSocket

The PeerSocket class is a implementation of the IPeerCommunicator interface, defining the methods with socket[5] communication. In addition to the methods defined in the IPeerCommunicator interface, there is a *testConnection()* method that tries to connect to the specified host, and returns the result, success or failure, of the test.

[5] A socket is a communication model that uses IP to transfer data between two processes.

PeerSocket contains three configuration parameters for the socket. These are *SOCKET_TIMEOUT*, *SOCKET_OPEN_TIMEOUT* and *SOCKET_WRITE_SLEEP*. *SOCKET_TIMEOUT* defines the time a socket is kept open before close, meaning that if the socket has no activity for that particular amount of time, the connection will tear down. The default parameter is 30 seconds. Any activity after the socket has closed will result in a reconnect, which is performed seamless to rest of the application. The *SOCKET_OPEN_TIMEOUT* parameter states the timeout when opening a socket to a remote host, default is 2 seconds. The timeout only applies to the stage where the socket is trying to open a connection. If the remote host does not respond within that time, the connection is interrupted. For instance, when communicating over a slow connections, the parameter could be configured higher. The last parameter, *SOCKET_WRITE_SLEEP*, states how long the socket shall wait after each write operation. The default value is 200 milliseconds. The *SOCKET_WRITE_SLEEP* parameter is included because of an implementation issue, and is discussed in Section 5.2.

In TLS, the socket is closed after the application data has been transmitted. When new application data is to be sent, a new socket is connected and a session resume is conducted. In EduTLS, a variant has been designed, where the socket stays open for *SOCKET_TIMEOUT* seconds before closing. If more application data is sent before timeout, the timeout gets reset to its initial value. If the timeout expires, and more application data is to be sent, the socket must reconnect and a session resume must be conducted.

Table 4.4: The server package source code

| **Class name** | **Size** |
|---|---|
| IPeerCommunicator | 23 lines of codes (12 without comments) |
| Listener | 186 lines of codes (146 without comments) |
| PeerSocket | 220 lines of codes (164 without comments) |

### 4.4.3 The Common Package

The common package contains functionality that is used by every part of the system.

#### 4.4.3.1 Log

Log and LogEvent provides the logging service, where the whole system may contribute. LogEvent describes one single log event, with a title and description. The Log class is a collection holder of log events. The graphical user interface is a log subscriber, making it receive every log event that is created.

Log events are organized hierarchical to make it easier to browse. For example, when a handshake occurs, a handshake log event is created, and every message sent during that handshake is created as a log event child of the handshake log event.

#### 4.4.3.2 Tools

The Tools class is also a part of the common package. This class contains various methods for text and byte manipulation and conversion. An example of a method in the Tools class is *compareByteArray*. This method takes two arguments, both of type byte array, and compares every byte in each array to examine if they are equal. The method is demonstrated in the listing below.

```
1  public static boolean compareByteArray(byte[] one, byte[] two) {
2     /* Not equal length, hence they cannot be equal */
3     if(one.length != two.length)
4        return false;
5     for(int i = 0; i < one.length; i++) {
6     /* loop through every byte and compare */
7        if(one[i] != two[i])
8           return false;
9        }
10    /* they are equal */
11    return true;
12    }
```

Listing 4.14: The compareByteArray() Method

Table 4.5: The common package source code

| Class name | Size |
|---|---|
| Log | 32 lines of codes (19 without comments) |
| LogEvent | 92 lines of codes (52 without comments) |
| Tools | 211 lines of codes (122 without comments) |

### 4.4.4 The Crypto Package

The crypto package consists of the PRF class and five interfaces; ICipher, ICompression, IHash, IKeyExchange, and IRandomGen. The PRF class is a pseudo random function, used by TLS to produce the master secret and the key block. According to the specifications of TLSv1.2, the PRF function is cipher suite dependent[10]. In this implementation, however, all cipher suites uses the same PRF function, identical to the TLSv1.1 specification [12].

The five interfaces describe the requirements for each of the succeeding chapters.

#### 4.4.4.1 The Crypto.Cipher Package

The classes in the crypto.cipher package must implement the interface ICipher. The following methods are defined in ICipher:

```
1 public interface ICipher {
2    public int getBlockSize();
3    public int getKeySize();
4    public void cipher(byte[] input, int inOff, byte[] output, int outOff);
5    public String getName();
6    public void init(boolean forEncryption, byte[] key);
7    }
```

Listing 4.15: The ICipher Interface

There are four classes in the cipher package. These are None, Rijndael, Rijndael2 and DES. None is a empty cipher, used in the session state before an actual cipher suite is negotiated.

The Rijndael implementation in the Rijndael class is taken from the Bouncy Castle project[20], and the Rijndael2 class is taken from the GNU Crypto project[21]. There have been done slightly modification of the classes to make them satisfy the ICipher interface. Two different implementation of the same algorithm was included to demonstrate performance variances, see Figure 4.7 for more information. The last cipher algorithm is DES, where the implementation is taken from Dr. Herong Yang's cryptography tutorials[38].

#### 4.4.4.2 The Crypto.Compression Package

Compression is the process of reducing the size of a bit-sequence, by exploiting statistical redundancy to represent equal parts of the bit sequence with a short reference.

```
1 public interface ICompression {
2    public byte[] compress(byte[] input);
3    public byte[] decompress(byte[] input);
4    public byte getCompressionId();
5    public boolean isEnabled();
6    public void setEnabled(boolean enabled);
7    public String getName();
8    }
```

Listing 4.16: The ICompression Interface

There is one compression algorithm implemented in addition to *CompressionMethod.null*, which is the ZLib algorithm. ZLib is a part of the Java native library, located in the Deflater and Inflater classes. EduTLS uses these native libraries, the complete source code of the compression algorithm is thus not included.

The compression interface has specified the methods *isEnabled*, *setEnabled*, and *getCompressionId*. The reason is that the compression method is not a part of the cipher suite, and these methods make it possible to turn on and off the support for a certain compression method, and for two communicating peers to agree on the same algorithm it is better to use an identifier than the algorithm name.

#### 4.4.4.3 The Crypto.Hash Package

The Hash package contains all the cryptographic hash algorithms, used both to calculate hash values, message authentication codes, and in the PRF generation.

```
1 public interface IHash {
2    public byte[] getHash(byte[] input);
3    public int getSize();
4    public String getName();
5 }
```

Listing 4.17: The IHash Interface

There are two hash functions implemented, the SHA-1 and SHA-256. The interface only has defined three functions, as specified in the listing above. In addition to the hash calculation, the hash functions are used to create the HMAC digest. The HMAC is calculated with the procedures described in Section 3.6.

#### 4.4.4.4 The Crypto.KeyExchange Package

```
1  public interface IKeyExchange {
2     public BigInteger getPublicKey();
3     public BigInteger getPublicModulus();
4     public BigInteger getSecretKey();
5     public boolean requireServerKeyExchange();
6     public BigInteger getServerKeyExchangeMessage();
7     public String getName();
8     public void initKeys(int size);
9     public void setYb(BigInteger yb);
10 }
```

Listing 4.18: The IKeyExchange Interface

Two classes implement the IKeyExchange interface; DH and RSA. The former is an implementation of the Diffie-Hellman key exchange algorithm, whereas the latter is the Rivest-Shamir-Adleman algorithm.

Note that the methods defined in the IKeyExchange interface only deals with key exchange. The RSA algorithm is also suitable for encryption and signature, but this is not part of the key exchange and not specified in the interface.

The *setYb* method is used by the Diffie-Hellman algorithm to calculate the secret key from the public key of the remote peer. The method is not used by the RSA algorithm.

The *requireServerKeyExchange* returns whether the algorithms requires a message in addition to the server certificate. This is true for the Diffie-Hellman algorithm, because the base generator must be exchanged in addition to the public key and modulus. For the RSA algorithm, the server key exchange is not required. The handshake protocol calls the *requireServerKeyExchange* of the negotiated key exchange algorithm, and if it responds true, the handshake sends the server key exchange message, with the content from the *getServerKeyExchangeMessage* of the algorithm.

The *getSecretKey* method returns the calculated secret key in the Diffie-Hellman algorithm, and the private exponent in the RSA algorithm.

#### 4.4.4.5 The Crypto.Random Package

The Random package contains the pseudorandom bytes generators. These are used to generate random byte streams of a specified size, used in the handshake.

```
1 public interface IRandomGen {
2     public byte[] randBytes(int nbytes);
3     public String getName();
4 }
```

Listing 4.19: The IRandomGen Interface

The only class in the Random package is BlumBlumShub, which is a complete implementation of the blum-blum-shub algorithm. The blum-blum-shub algorithm is explained in Section 2.2.4.1.

Table 4.6: The crypto package source code

| **Class name** | **Size** |
|---|---|
| Log | 32 lines of codes (19 without comments) |
| ICipher | 19 lines of codes (10 without comments) |
| ICompression | 19 lines of codes (11 without comments) |
| IHash | 10 lines of codes (8 without comments) |
| IKeyExchange | 17 lines of codes (15 without comments) |
| IRandomGen | 6 lines of codes (5 without comments) |
| PRF | 77 lines of codes (57 without comments) |
| cipher.DES | 278 lines of codes (258 without comments) |
| cipher.None | 32 lines of codes (23 without comments) |
| cipher.Rijndael | 716 lines of codes (556 without comments) |
| cipher.Rijndael2 | 822 lines of codes (608 without comments) |
| compression.None | 38 lines of codes (28 without comments) |
| compression.ZLib | 75 lines of codes (60 without comments) |
| hash.None | 20 lines of codes (15 without comments) |
| hash.SHA1 | 239 lines of codes (156 without comments) |
| hash.SHA256 | 164 lines of codes (112 without comments) |
| keyexchange.DH | 80 lines of codes (58 without comments) |
| keyexchange.None | 79 lines of codes (64 without comments) |
| keyexchange.RSA | 180 lines of codes (109 without comments) |
| random.BlumBlumShub | 181 lines of codes (138 without comments) |

### 4.4.5 The TLS Package

The TLS package contains the ”core” of the TLS protocol. This includes the engine, the handshake protocol, the record layer, the alert protocol with its alert exception class, connection state definitions, certificate authority, cipher suite definition, and the IApplication interface.

The TLSEngine is the communication link between the application and the TLS protocol. This involves session management, remote peer connectivity, and sending and receiving of messages. The TLSEngine has the same responsibility as javax.net.ssl.SSLSocketFactory, which is the native implementation of the SSL/TLS protocol in Java. Instead of using a SSLSocket, EduTLS communicates through a regular and insecure socket connection, and uses the TLSEngine to ensure communication privacy. The TLSEngine has no important functions by itself, it is just a coordinator that relies on functionality from the other components.

The CertificateAuthority class demonstrates a simple PKI in EduTLS. It provides functionality to sign and verify certificates. CertificateAuthority has a built-in an RSA key-pair used in the signing and verification process. In a real PKI infrastructure, the certificate authority would have an independent role, not included as a part of the implementation like in EduTLS. It has been included to offer the signing and verification features.

The AlertException class is a subclass of java.lang.Exception, meaning it can generate an error if a certain condition is not satisfied. The error has a code and a description, and the application can capture and treat the error in a reasonable way, instead of resulting in a program crash. An AlertException error is generated in EduTLS when a failure in the communication occurs. The majority of points where these failures can occur are during a handshake, for instance if no mutual cipher suite is supported. If an error occurs, the session is destroyed. A list of the error codes is given in Appendix A.2.

The CipherSuite class defines a cipher suite. This includes an encryption algorithm, key exchange algorithm, hash function, and the name and code for the cipher suite, and a flag indicating if the cipher suite is enabled or not. The TLSEngine contains a list of all supported cipher suites. It is not possible for the user to add new cipher suites at runtime, only enable or disable the predefined ones. It is, however, easy to create new cipher suites in the source code, and it is supported by the application automatically.

The State class holds session information for one connection, called the connection state. The parameters in a connection state is listed in Table 4.7.

Table 4.7: Connection State

| Parameter | Description |
|---|---|
| changeCipherSpecClient | A true or false condition whether the client has sent the change cipher spec message |
| changeCipherSpecServer | A true or false condition whether the server has sent the change cipher spec message |
| cipherSuite | The current active cipher suite for this connection |
| client_write_encryption_key | The cipher algorithm encryption key for the client |
| client_write_IV | The cipher algorithm IV for the client |
| client_write_MAC_key | The hash function key of the MAC for the client |
| clientRandom | The client random |
| compressionMethod | The compression method in use for this connection |
| isResumableSession | A true or false condition whether this session is resumable or not |
| masterSecret | The master secret |
| peer | The peer belonging to this connection. Only used for determining the connection end and host address |
| preMasterSecret | The pre-master secret |
| server_write_encryption_key | The cipher algorithm encryption key for the server |
| server_write_IV | The cipher algorithm IV for the server |
| server_write_MAC_key | The hash function key of the MAC for the server |
| serverRandom | The server random |
| sessionId | The session identifier for this connection |

TLSEngine has a list of all connection states, and the current active state. When a new connection is established, the engine examines for previous connection states to that particular host. If it is found, the handshake performs a session resume, otherwise a new connection state is established. Note that the *isResumableSession* parameter is always true in EduTLS, hence a session is always resumed if an old connection state exists.

#### 4.4.5.1 The TLS.Handshake Package

The handshake package contains all messages sent during a full handshake. In addition to the handshake messages, there is the IHandshakeMessage interface, defining the requirements for a handshake message.

```
1 public interface IHandshakeMessage {
2    public byte[] getByte();
3    public byte getType();
4    public String getStringValue();
5    public String toString();
6 }
```

Listing 4.20: The IHandshakeMessage Interface

The *getByte* method returns the byte value of the handshake message, which is sent to the remote peer. The *getStringValue* returns the human readable representation of the byte value, used by the log service. The *getType* returns the handshake message type, and *toString* returns the human readable representation of the type.

#### 4.4.5.2 The TLS.Record Package

The record package contains the objects significant to the Record protocol. This includes the TLSCiphertext, TLSCompressed and TLSPlaintext. When a TLSRecord object is created, it transform the text to TLSPlaintext, which is just a fragmentation of the text to chunks of $2^{14}$ bytes or less. Each of the TLSPlaintext objects is then converted to TLSCompressed by using the compression method specified in the connection state. This is followed by adding the message authentication code with the hash algorithm of the active cipher suite, also specified in the connection state. Each of the TLSCompressed objects is then converted to TLSCiphertext by encrypting the object with the cipher algorithm and the calculated key, both specified in the connection state. In each of these transformations, a new header is added with the appropriate parameters. The most important parameter is the length, since the size of the object often varies between each transformation[6].

When a TLSRecord object is received from the remote peer, the same procedures are performed, but in reverse order.

Table 4.8: The TLS package source code

| **Class name** | **Size** |
|---|---|
| AlertException | 90 lines of codes (55 without comments) |
| CertificateAuthority | 45 lines of codes (36 without comments) |
| CipherSuite | 66 lines of codes (45 without comments) |
| ConnectionStates | 34 lines of codes (25 without comments) |
| IApplication | 7 lines of codes (6 without comments) |
| State | 262 lines of codes (202 without comments) |
| TLSAlert | 36 lines of codes (28 without comments) |
| TLSEngine | 283 lines of codes (172 without comments) |
| TLSHandshake | 345 lines of codes (275 without comments) |
| TLSRecord | 288 lines of codes (219 without comments) |
| handshake.ChangeCipherSpec | 27 lines of codes (20 without comments) |
| handshake.ClientHello | 157 lines of codes (140 without comments) |
| handshake.ClientKeyExchange | 93 lines of codes (73 without comments) |
| handshake.Finished | 44 lines of codes (31 without comments) |
| handshake.IHandshakeMessage | 10 lines of codes (8 without comments) |
| handshake.ServerCertificate | 178 lines of codes (149 without comments) |
| handshake.ServerHello | 152 lines of codes (126 without comments) |
| handshake.ServerHelloDone | 27 lines of codes (20 without comments) |
| handshake.ServerKeyExchange | 40 lines of codes (30 without comments) |
| record.TLSCiphertext | 122 lines of codes (105 without comments) |
| record.TLSCompressed | 54 lines of codes (43 without comments) |
| record.TLSPlaintext | 49 lines of codes (28 without comments) |

[6]TLSPlaintext to TLSCompressed may compress and decrease the size, and TLSCompressed to TLSCiphertext may increase the size because of padding

## 4.5 The Architecture

### 4.5.1 Applied Patterns

This section describes how the architecture of the system is designed. The implementation uses both architectural patterns and design patterns to achieve the requirements specified in Section 4.1. A pattern is a well-proven and tested, general solution to a commonly occurring problem. [33]

#### 4.5.1.1 Model-View-Controller

An important and popular pattern is the Model-View-Controller (MVC) pattern. The MVC pattern isolates the application logic from the user interface to achieve separation of concerns, making the source code easier to maintain and familiarize with. Another feature of MVC is the ability to use multiple views at the same model, which also makes it easier to create automated tests for the application. MVC consists of three separate parts; [33]

**Model** encapsulates the core data for the application

**View** the visual representation of that data to the user

**Controller** receives input, usually from the view, translates it to service requests, and send it to the model or the view

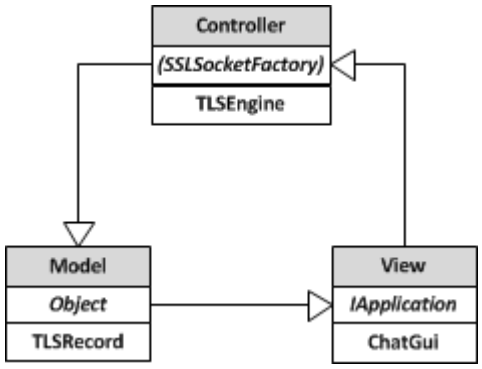


Figure 4.9: Model-View-Controller pattern

Figure 4.5.1.1 shows how the MVC pattern has been implemented in EduTLS. By using this separation, it is be easy to implement a new user interface by replacing the view. This abstraction is similar to TLS, where any application can use TLS in the communication.

Note that EduTLS implements the interface IApplication. TLSEngine is an independent class, but if this project was a complete implementation of SSL/TLS, the controller component would subclass the SSLSocketFactory class. TLSRecord is a self-defined object, containing all relevant information.

#### 4.5.1.2 Client-Server

Just like in TLS, the application is built on a client-server model. EduTLS acts both like a server and a client, where the user interaction decides the role of the system. The client makes the initial contact by sending a request to an external server. The server responds, and if the communication channel was successfully established, the two peers can send messages to each other. At this point, they are equals. [33]

#### 4.5.1.3 Observer

In this pattern there are two roles, subject and observer, where an observer subscribes to be notified when a subject's state changes. The great advantage here is that the observers will be notified automatically instead of having to ask the subject at regular intervals whether there are any changes. Another advantage is that the observers can decide who they want to subscribe to, and at what time they want to subscribe. [33]

The Log act as a subject and ChatGui is the observer. Any part of the application can add log events to the Log component, which distributes the messages forward to all of its observers. In combination with the MVC model, this creates a better separation of the logic and the user interface.

#### 4.5.1.4 Singleton

The Singleton pattern is used to restrict the instantiation of a class to one object. This is useful when the application only needs one instance of the object and it is shared among the other components of the system. The Log class is implemented as a Singleton class, because there should be only one instance of the class, and that instance is needed across the system. [33]

## 4.6 Automated Tests

One of the goals with this application has been to make it manageable for others to extend or modify it. Of that reason, it was decided to create a set of automatic tests. Introduction to automated testing is given in Section 2.4.2.

The tests in EduTLS are made with JUnit[7] version 4. EduTLS only contains unit tests and integration tests. The functional tests have been performed manually, due to the complexity of a functional test framework compared to the basic user interface in this application.

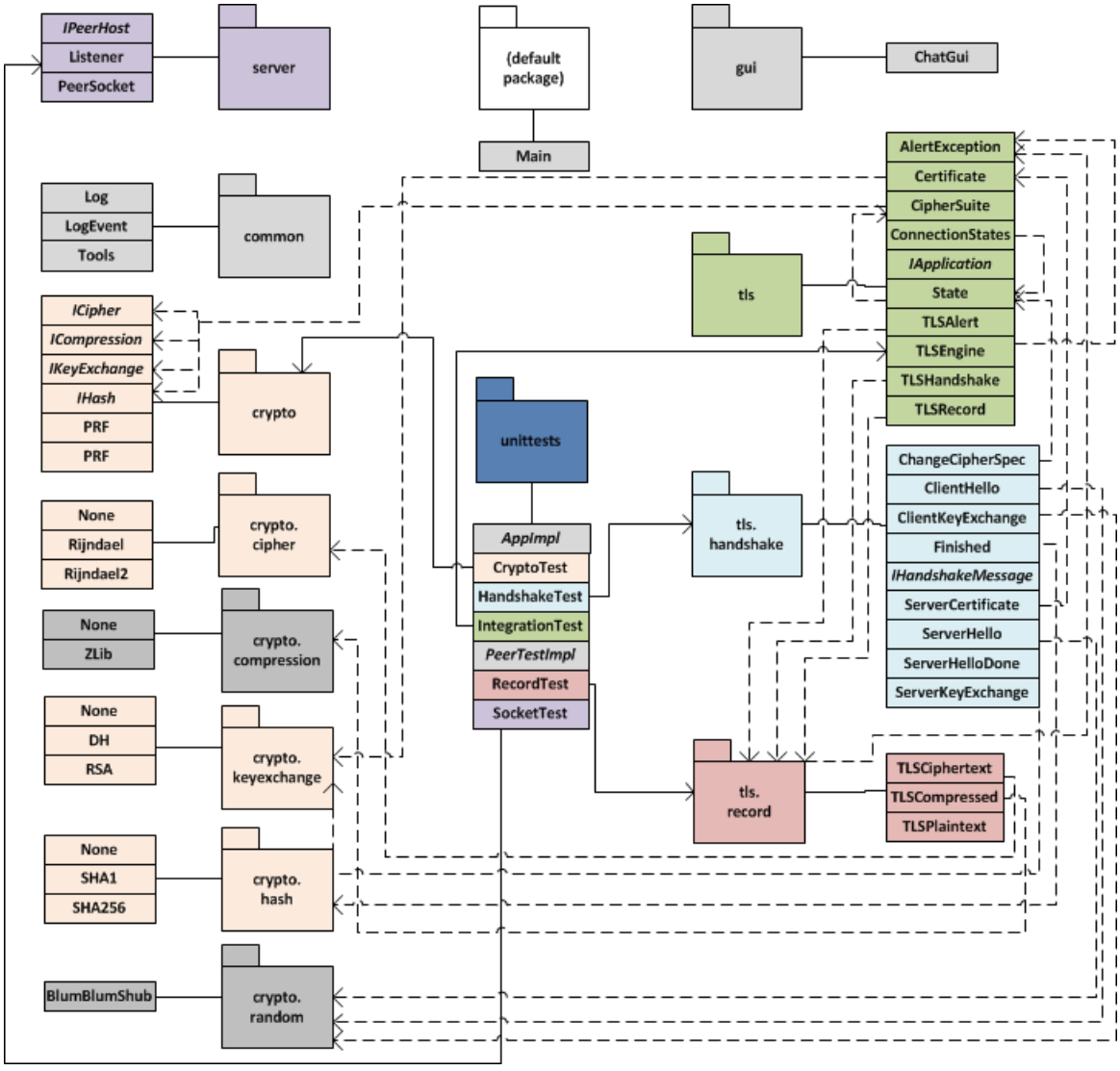


Figure 4.10: Test coverage

Figure 4.6 visualizes what classes the different tests cover. The solid lines show which classes the test validates, while the dotted lines show component dependencies. Recall Section 2.4.2, that when creating unit tests one component should be isolated from rest of the system when tested.

[7] http://www.junit.org -the most popular test framework for Java

In a commercial application, the tests should cover the entire application. That is not the case for EduTLS, classes in gray are not tested. These classes have been manually tested, because they are not vital parts of the protocol.

#### 4.6.0.5 AppImpl

AppImpl is a moch implementation of the IApplication interface. A description of IApplication is given in Section 4.4.5. The TLSEngine class requires a class implementing the IApplication interface, therefore all test classes that utilize the TLSEngine creates an instance of the AppImpl. The only practical function of AppImpl is to keep track of the last message it receives.

#### 4.6.0.6 CryptoTest

CryptoTest has seven methods; *testSha1()*, *testSha256()*, *testRijndael()*, *testRijndael2()*, *testDes()*, *testDH()* and *testRSA()*.

The first two methods test the SHA1 class and SHA256 class by calculating the digest value of three different text values and compare it with predefined values. The predefined values were calculated with a known-correct implementation of the hash function. See Appendix B.3 for a details of the calculation of predefined values.

The methods *testRijndael* and *testRijndael2* tests the two Rijndael cipher implementations. This is accomplished by the same principle as with the digest test; it compares calculations with predefined values that are known to be correct. The *testDes* verifies the DES implementation with the same procedures as in the Rijndael algorithm testing.

The *testDH()* method creates two instances of the DH class, exchange the public keys between them, and asserts that the secret key which is calculated by the two instances are equal. This is a simulation of a key exchange scenario.

The last method, *testRSA()* creates a key pair, and with that key pair it encrypts and decrypts a secret message. The encrypted and decrypted values are compared to be sure the encryption has been successful, and finally it asserts that the decrypted value is equal to the original message.

#### 4.6.0.7 HandshakeTest

HandshakeTest tests different part of the handshake. The method *testClientHello()* creates a ClientHello and asserts its correctness. The *testServerHello()* and *testCertificate()* have the same purpose with the ServerHello class and ServerCertificate, respectively. The *testClientHandshake()* and *testServerHandshake()* tests the TLSHandshake class. The *testCertificate* creates a certificate, and with the byte value of that certificate, it creates a new certificate, and compares these two certificates. The *testCertificateAuthority* creates a certificate which it signs, and verifies the signature of that certificate.

#### 4.6.0.8 IntegrationTest

IntegrationTest is responsible for testing the TLSEngine class. This is performed by initiating a connection and sending a message. The connection is established with a mock instance of PeerTestImpl, and the sending is validated with a mock instance of the AppImpl class. If the sending is success, and gets received, the test assumes that the components cooperates as they are suppose to.

#### 4.6.0.9 PeerTestImpl

PeerTestImpl is a moch implementation of the IPeerSocket interface, used only by the tests described in this section. The write method puts the TLSRecord object in a queue. When the read method is called, the first object in the queue is returned.

#### 4.6.0.10 RecordTest

RecordTest has two methods; *testErrorCodes()* and *testCipher()*. The former verifies that the record layer has sufficient error handling by feeding it with various illegal composed messages. The latter is more an integration test than a unit test. It creates a cipher suite, and uses the TLSRecord object to create the ciphertext of a large text sample. This ciphertext is then sent to a new TLSRecord object, provided with the same cipher suite, used to bring back the original text.

#### 4.6.0.11 SocketTest

The final test, SocketTest, tests the Listener class. After starting the listener service, it performs two tests. The first is a stress test with 100 connections to verify it can handle many simultaneous requests. The second test opens a connection and feed both legal and illegal values to verify that the listener service treat the requests like it is suppose to. There are three possible outcome from a request; a test request should respond with a successful test message, a TLS request should be forwarded to the TLSEngine for further processing, and any other request should result in a alert message.

# Chapter 5

# Discussion

*"Tell me, I'll forget. Show me, I'll remember. Involve me, I'll understand."*
- Chinese proverb

This chapter reflects the work that has been conducted in this thesis. The requirements specified in Table 4.1 gets special attention in the discussion, since they were the goals for the project. The next section presents the author's thoughts about the security of TLS, followed by a section of the challenges faced during implementation, before concluding the chapter with ideas for future improvements.

The source code has approximately 8000 lines of code when including the unit tests, and almost 6000 when stripping away all comments and empty lines. At first sight, it may seem like the project has become too complex for a student to easily familiarize with. However, the cryptographic primitives constitute about 44% of the source code, and these are not essential to go deeper into. The unit tests consists of approximately 800 lines of code, but neither of these methods are important to study when looking at the system.

Table 4.1 states 20 requirements. All the requirements are measurable, with exception of the first requirement; "simplify - the application shall make it easier to understand TLS". The only way to conclude if the application makes it easier to understand TLS, is to conduct a survey or user tests. It would certainly give valuable feedback, but there has been no capacity or time to carry out such tests. The implementation was finish only a few weeks before deadline, and to conclude the success or failure of the goal, it must have been done at the end of the project.

The application is developed in Java, satisfying the second requirement. The third requirement states that the architecture shall be similar to TLS. The architecture has been thoroughly planned while studying the TLS specifications, and explained in Figure 4.1, Figure 4.4 and Section 4.5. The requirement must be concluded as fulfilled.

The handshake steps are identical to the TLS specification, and with the same content in each message. The handshake also supports session resume, where only two of the handshake steps are carried through at each peer. The handshake is explained in Section 4.3.3 and 4.4.5.1. The record protocol, discussed in Section 4.4.5.2, has the same responsibility as in TLS. That is, every object sent to the remote peer must be of type TLSRecord,

which means that the record protocol encapsulates its header information and processes the package before transmitting it. This satisfies requirement 4, 5 and 6.

The graphical user interface has a hierarchical log tree, where every log event has a time-stamp. The time-stamp gives the user information about how time consuming the operation has been, by comparing it to the previous log event. The GUI also has a performance testing tool, explained at Figure 4.7. The graphical user interface fulfills requirement 7 and 20.

In the crypto package, explained in Section 4.4.4, there are implementations of Diffie-Hellman, Rijndael, DES, SHA-1, SHA2-56, RSA with Miller-Rabin primality test, and the Blum-Blum-Shub random number generator. In addition, the crypto package has the ZLib compression method, but this is not an complete implementation, it uses the native Java library. These implementations meet requirements 11-18, but partly requirement 19 since it is not a complete implementation. The requirement 11, 12, 13 and 16 were actually fulfilled before starting the implementation, because they were already implemented in the EduSSL[3] project, and copied with only minor changes.

Requirement 8, automated tests, are partly satisfied. The application has several unit tests, and integration tests, but it would be of great advantage to have more tests, especially if someone else wants to modify or continue the project. The tests covers the cryptographic primitives, the record protocol, and most of the handshake. The handshake part should have more tests, since it is the most complex part of TLS.

## 5.1 Security Analysis

The Secure Socket Layer protocol has been publicly available for more than 17 years. When it was released, it became very popular in a short period of time. Like most technological inventions which receive much attention, it has been widely analyzed. Discovered weaknesses has resulted in new versions and continuous improvements of the protocol. Performing yet another security analysis of the protocol, as a part of a larger project with other goals, would unlikely result in any new findings. Hence, this chapter's focus is not a thorough security analysis, but rather a few thoughts from the author.

There are several ways of doing a security analysis. Most analyses of cryptographic protocol focuses on an abstract study of the protocol, but it is also important to analyze implementations. TLS is proven to be highly secure with the assumption that the chosen cipher suite is secure[47]. With the strong mandatory cipher suites specified in TLS version 1.2, it is undoubtedly a difficult protocol to break. The majority of flaws when using the TLS protocol is due to either implementation errors or user decisions. The implementation errors comes from how the developer has implemented the protocol, or how the protocol is used. When a developer uses a security protocol, a good understanding of the protocol is very important. Without this understanding, the developer might use the protocol in a situation it is not intended for, or under wrong assumptions. [47, 32]

The user decision errors reflect balance between a strict implementation or user involvement, regarding the certificate validation process. The TLS specification states that the certificate must be X.509v3, unless explicitly negotiated otherwise. It does, however,

not specify how the protocol should verify the certificate, and how to react if it fails. There are many areas where the verification might fail; unknown certificate authority, the validity period, signature failure, unsupported algorithm, conflicting common name, to mention the most common. In practice, this issue is often solved by involving the user. The majority of users have little, or no, knowledge on the consequences of the decision. [49, 50, 51]

The public key infrastructure solves the authentication process in TLS. During the handshake, both parties optionally exchange their certificate. The authentication of a certificate is a trusted third party, which verifies the owner and signs the certificate. These trusted third parties are organizations, called a certificate authority (CA), and is organized hierarchically. An important issue is how to deal with a CA that is compromised. The CA has possible issued thousands, or millions, of signatures, and if the trust in this particular CA is completely removed, it might result in serious consequences. This challenge is not a part of the TLS protocol, but it certainly an issue to consider in future version of TLS and its public key infrastructure. The Tor Project[1] concludes that in practice, the certificate revocation does not work, and particularly the situation if a CA gets compromised. [52]

Finally, it is worth mentioning that even after years of analyses and research, new weaknesses are discovered. In 2009, Marsh Ray published a new vulnerability, allowing an attacker to inject chosen plaintext into an encrypted data stream. The vulnerability is in the renegotiation process, and affects all application running on top of TLS. This attack reminded the community that new sophisticated approaches may arise, even after so many years. [53]

## 5.2 Implementation Challenges

This chapter presents the problems faced before and during the implementation of EduTLS, and how they were solved.

The majority of the implementation problems were due to unpredictable behavior in a couple of the native Java classes. During the handshake, several TLSRecord objects are sent consecutively. At the remote peer, it occasionally received more than one TLSRecord object in one read. To deal with the issue, it resulted in a lot of testing and error handling, making the communication system hard to familiarize with. Instead, a decision was made to have a more relaxed communication model, where the communication pauses for 200 milliseconds after each time it sends a TLSRecord object to the remote peer. This resulted in a performance penalty, but simplicity has higher priority than performance in this project.

The application was developed on a machine with the Windows operating system. Java is system independent, and runs on all operating systems that have the JVM installed. When testing communication between a Windows system and a Linux system, the handshake

[1] Tor is free software and an open network that helps you defend against a form of network surveillance that threatens personal freedom and privacy, confidential business activities and relationships, and state security known as traffic analysis

failed when the server sent its certificate. The reason turned out to be because the certificate is ascii encoded and contained a line separation character[2] that is specific to the operating system. When replacing this with a system independent character[3] the problem was solved. This is one reason why automated testing by itself is not sufficient.

The handshake was designed and planned well before implementation. Besides the problem with more than one TLSRecord object in one read, it turned out to be very well organized. The record protocol however, was not prioritized highly enough in the design process. The overall approach was planned, with every step in the process according to the TLS version 1.2 RFC, but at the last stage of the implementation, a few problems were encountered. The problem was mainly due to lack of automated tests in the transformation between TLSPlaintext and TLSCiphertext objects. This made the debugging hard, and even though the problems were solved, more unit tests should have been created. This is discussed more in the next section.

## 5.3 Further Improvements

This section discusses improvement areas for the application.

When looking back at the project, there has been both wise and not-so-wise decisions. The most affecting design decision was probably that the graphical user interface was not planned well enough in the initial design phase. The performance test could have been better if it was taken into account when creating the cryptographic primitives, but the performance test was the lowest prioritized requirement and therefore it did not get much attention in the beginning. Performance is a very important topic in cryptography, and the tests could beneficially give more detailed measures, give the user ability to choose between the primitives to test, and calculate the ratio for compression algorithms. The pseudorandom generators could have been included in the performance test. The GUI could also give the user opportunity to browse sessions and look at the information about the session; chosen cipher suite, last connection, duration, when it expires and so on.

One goal for future work is to make the protocol compatible with the TLS protocol. The EduTLS application has been designed to minimize the effort required to accomplish this goal. To make the protocol comply the TLS specification is a tremendous amount of work, but it could certainly make it compatible for testing purposes, without satisfying all requirements of the specification.

The security of the EduTLS protocol has room for several improvements. For example, there is no message forgery detection implemented. To implement this security goal, one would need to add sequence numbers to the connection state, and include the sequence number when calculating the message authentication mode for each message.

The graphical user interface shows how time consuming every operation is, but could additionally include a summary of which operation was most costly, and perhaps a more fine grained time measure inside each operation.

[2] System.getProperty(”line.separator”)
[3] ”\n”

The certificate authority and certificate can be solved more smoothly. The CA serves the purpose of signing and verifying certificates, but to demonstrate a more realistic public key infrastructure, there could have been a hierarchical structure of several certificate authorities, and let the user choose which one to trust and to sign its certificate. This would, however, complicate and expand the application significantly, but could be an idea for future improvements. Another task could be to utilize X.509 certificates instead of the built-in certificate and certificate authority. This topic was under consideration in the design process, but was discarded due to increased complexity of the system. However, the experience from such an implementation would certainly be valuable.

## 5.4 Related Work

To the author known, there is no other project with an academic security protocol implemented. Several SSL and TLS implementations exist, but these are complete implementations of the protocol. One of the most complex is Jessie[54], which is a complete implementation of a provider[4], The source code of Jessie consists of over 40.000 lines of code, which does not satisfy the first requirement *The application shall make it easier to understand TLS*. Another project is the Bouncy Castle[20], which is a Java implementation of SSLv3 and TLSv1, but this project also fails to satisfy the first requirement of simplicity.

[4] A provider is a system that implements some or all parts of the Java Security (cryptographic services)

Chapter 6

# Conclusion

*"Knowledge is a treasure that will follow its owner everywhere."*
- Chinese proverb

Internet is a result of a research project called ARPANET, created by the US military and the US Department of Defense during the Cold War. When designed, the goal was to create a decentralized network with no single node in charge. This property made the network continue to operate even after attacks on several nodes; a great advantage in a war. Since the users knew and trusted each other, security amongst them was not an issue. In the beginning of the 1980s, the number of nodes that were connected increased exponential, and the need for secure communication appeared. One of the responses was from Netscape Communications, which released the Secure Socket Layer (SSL) protocol in 1994. Because of its design, the protocol gained a lot of attention, and was adopted as an Internet standard in 1999, and called Transport Layer Security (TLS). Even after a decade, TLS is still the predominant protocol for secure network communication. From being a protocol intended for the government and the military, it is now a part of most people's everyday lives. The result is an explosion in the demand for people with knowledge in information security.

The purpose of this thesis has been create a security application which can be used when studying information security. The main concern is the SSL/TLS protocol, a cryptographic system consisting of more than 30 cryptographic primitives and protocols. The application contains an implementation of protocol similar to, but less complex than, TLS, and a graphical user interface. The simplified TLS protocol contains most of the operations and parts of the TLS protocol, and several cryptographic primitives. The architecture, the record layer, and the handshake protocol is the most complex parts of TLS, hence there has been most attention on these in the project. The graphical user interface presents every operation of the protocol, with detailed information and how time consuming it has been. The result involves both the source code and the application, where the idea is to study one part, or preferably both, to get a practical aspect of a cryptographic system.

The application is designed for modification to minimize the effort required to expand the system. This involves, but is not limited to, adding new cryptographic primitives and

cipher suites, enhance the graphical user interface, broaden the logging service, or add more security to the protocol.

The challenges I have encountered during the implementation is mention in Section 5.2. The biggest challenge, however, has been to find good literature on how to accomplish its own TLS implementation. This is probably because it is not intended to do so, but rather use existing and well tested implementation, that every programming language should have in its native library. Experiencing this has given me even more motivation in this project, believing that my work could help other in the same situation.

The main objective in this project is to emphasize the importance of using the technology as it is intended to be used. Correct use is related to understanding, and this project offers a practical approach to learn and understand the most important cryptographic system; TLS.

# Bibliography


[1] *On the Robustness of Applications Based on the SSL and TLS Security Protocols*
Diana Berbecaru and Antonio Lioy
Springer-Verlag Berlin Heidelberg, 2007

[2] *Handbook of Applied Cryptography*
A. Menezes, P. van Oorschot, and S. Vanstone
CRC Press, 1996

[3] *Educational implementation of SSL/TLS*
Eivind Vinje
IME Faculty, NTNU
Specialization Project, Fall 2010

[4] *The Internet - past, present and future*
S P Sim and S Rudkin
BT Technol J Vol 15 No 2 April 1997

[5] *The Multimedia Internet*
Stephen Weinstein
Springer, 2005

[6] *A Brief History of the Internet*
Tania Regina Tronco
Campinas
Sao Paulo, CEP 13096-902, Brazil

[7] *Cryptography and Network Security*
Fourth Edition
William Stallings
Pearson International Edition

[8] *A Brief History of Cryptography*
Cypher Research Laboratories
Last modified: January 24, 2006
Last accessed: April 20, 2011

[9] *Java(TM) Language Specification, Third Edition*

J. Gosling, B. Joy, G. Steele, and G. Bracha
Publisher: Addison Wesley; 3 edition (June 24, 2005)
ISBN-13: 978-0321246783

[10] Request For Comment 5246
http://www.ietf.org/rfc/rfc5246.txt
The TLS Protocol Version 1.2

[11] Request For Comment 2246
http://www.ietf.org/rfc/rfc2246.txt
The TLS Protocol Version 1.0

[12] Request For Comment 4346
http://www.ietf.org/rfc/rfc4346.txt
The TLS Protocol Version 1.1

[13] *A Modular Security Analysis of the TLS Handshake Protocol*
P. Morrissey, N.P. Smart, and B. Warinschi
Advances in Cryptology
AsiaCrypt 2008, pp. 55–73

[14] *SSL and TLS: A Beginners Guide*
http://www.sans.org/reading_room/whitepapers/protocols/ssl-tls-beginners-guide_1029
SANS Institute InfoSec Reading Room
Last accessed April 20, 2011

[15] *Contemporary Cryptography*
Rolf Oppliger
ISBN 1-58053-642-5
Artech House, Computer Security Series

[16] http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html
National Institute of Standards and Technology
BLOCK CIPHER MODES
Last accessed May 21, 2011

[17] *Hash Function Combiners in TLS and SSL*
Marc Fischlin, Anja Lehmann, and Daniel Wagner
Darmstadt University of Technology, Germany

[18] *iSeries - Secure Sockets Layer*
http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/rzain/rzain.pdf
IBM
Version 5 Release 3

[19] *RFC3280 - Internet X.509*
*Public Key Infrastructure Certificate and*
*Certificate Revocation List (CRL) Profile*
http://www.ietf.org/rfc/rfc3280.txt

Last accessed: May 11, 2011

[20] *Legion of the Bouncy Castle*
http://www.bouncycastle.org/
Last accessed: May 11, 2011

[21] *The GNU Crypto project*
http://www.gnu.org/software/gnu-crypto/
Last accessed: May 11, 2011

[22] *Recommendation for Key Management*
National Institute of Standards and Technology
Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid
March, 2007

[23] *How CA Certificates Work*
http://technet.microsoft.com/en-us/library/cc737264(WS.10).aspx
Last modified: March 28, 2003
Last accessed: February 12, 2011

[24] *Understanding Digital Certificates*
http://technet.microsoft.com/en-us/library/bb123848(EXCHG.65).aspx
Last modified: March 19, 2005
Last accessed: February 12, 2011

[25] http://www.europki.org/europki/faqtroubl/en_index.html
*EuroPKI FAQ and troubleshooting*
Last modified: October 10, 2003
Last accessed: February 12, 2011

[26] http://en.wikipedia.org/wiki/OSI_model
*Wikipedia - Open Systems Interconnection model*
Last modified: May 4, 2011
Last accessed: May 5, 2011

[27] *How (Not) to Design RSA Signature Schemes*
Jean-Franqois Misarsky
France Telecom - Branche Developpement

[28] http://www.pgpi.org/doc/pgpintro/
*How PGP works*
Last accessed: January 29, 2011

[29] http://www.javabeat.net/tips/60-java-security-packages-using-jcajce.html
*Java Security Packages using JCA/JCE*
Last accessed: April 8, 2011

[30] http://www.15seconds.com/issue/991216.htm
*Crash Course in Cryptography*
Peter Persits
Last accessed: April 8, 2011

[31] http://www.praxiom.com/iso-27001-definitions.htm
ISO 27001 and ISO 27002
Last accessed: April 8, 2011

[32] *Cryptographic Protocol Analysis on Real C Code*
Jean Goubault-Larrecq
Fabrice Parrennes
LSV/CNRS UMR 8643 & INRIA Futurs projet SECSI & ENS Cachan
RATP, EST/ISF/QS LAC VC42
France

[33] *Pattern-oriented Software Architecture: A system of patterns*
Douglas C. Schmidt
John Wiley & Sons Ltd
ISBN 0 471 95869 7

[34] http://en.wikipedia.org/wiki/Cryptographic_primitive
*Wikipedia - Cryptographic primitive*
Last modified: November 14, 2009
Last accessed: March 2, 2011

[35] http://en.wikipedia.org/wiki/Cryptosystem
*Wikipedia - Cryptographic system*
Last modified: April 21, 2011
Last accessed: May 3, 2011

[36] http://fchabaud.free.fr/English/Publications/sha.pdf
Chabaud and Joux, 1998
Last accessed: May 21, 2011

[37] *Secure Hash Standard*
Federal Information Processing Standards (FIPS)
Publication 180-2
2002 August 1

[38] http://www.herongyang.com/crypto/des_implJava.html
Dr. Herong Yang, version 4.00
Cryptography Tutorials
Last accessed: May 19, 2011

[39] *Secure Hashing*
Cryptographic Toolkit
National Institute of Standards and Technology
Last modified: April 12, 2011

[40] *Digital Signatures*
Cryptographic Toolkit
National Institute of Standards and Technology
Last modified: Jan 30, 2006

[41] *Digital Signature Algorithm*

Wikipedia - the free Encyclopedia
http://en.wikipedia.org/wiki/Digital_Signature_Algorithm
Last modified: April 18, 2011
Last accessed: May 21, 2011

[42] *NOU 2001 : 10 Uten penn og blekk*
3. Hva er digitale signaturer og PKI?
Den Norske Regjering
Fornyings-, administrasjons- og kirkedepartementet
(only in Norwegian)

[43] *Introduction to Public Key Technology*
*and the Federal PKI Infrastructure* Cryptographic Applications & Protocols
National Institute of Standards and Technology
Last modified: February 26, 2001

[44] *Automated Test Generation and Verified Software*
John Rushby
Computer Science Laboratory
SRI International

[45] *Crypto++ 5.6.0 Benchmarks*
http://www.cryptopp.com/benchmarks.html
Last modified: March 31, 2009
Last accessed: June 02, 2011

[46] *WEB COMPONENT DEVELOPMENT WITH ZOPE 3*
Chapter 12 Automated Testing
Philipp Weitershausen
Springer
ISBN-10: 3540338071

[47] *A Standard-Model Security Analysis of TLS*
T. Jager, F. Kohlar, S. Schäge and J. Schwenk
Last modified: May 6, 2011
Last accessed: May 19, 2011

[48] *Analysis of the SSL 3.0 protocol*
D. Wagner and B. Schneier
USENIX Press, November 1996

[49] *Crying Wolf: An Empirical Study of SSL Warning Effectiveness*
J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor
Carnegie Mellon University
USENIX Press, April 2009

[50] *Hardening Web Browsers Against Man-in-the-Middle and Eavesdropping Attacks*
H. Xia and J. C. Brustoloni
Department of Computer Science, University of Pittsburgh
Published by ACM, 2005

[51] *The Emperor's New Security Indicators*
*An evaluation of website authentication and the effect of role playing on usability studies*
S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer
IEEE Symposium on Security and Privacy, 2007

[52] *The Tor Project*
Blog - Detecting Certificate Authority compromises and web browser collusion
https://blog.torproject.org/blog/detecting-certificate-authority-compromises-and-
Last updated: March 22, 2011
Last accessed May 21, 2011

[53] *Authentication Gap in TLS Renegotiation*
http://extendedsubset.com/?p=8
Marsh Ray
Last updated: November 5, 2009
Last accessed: May 21, 2011

[54] *Jessie: A free implementation of the JSSE*
http://www.nongnu.org/jessie/
Last accessed: May 11, 2011

Appendix A

# Tables and Lists

## A.1 The First Fifteen ARPANET Sites

* Bolt Baranek and Newman (BBN)

  Carnegie Mellon University

* Case Western Reserve University

* Harvard University

* Lincoln Laboratories

* Massachusetts Institute of Technology (MIT)

* NASA at AMES

* RAND Corporation

* Stanford Research Institute (SRI)

* Stanford University

* System Development Corporation

* University of California at Los Angeles (UCLA)

* University of California of Santa Barbara

* University of Illinois at Urbana

* University of UTAH

## A.2 TLS Alert Codes

Table A.1: TLS version 1.2 Alert Codes

| Code | Description |
|---|---|
| 0 | Close Notify |
| 10 | Unexpected Message |
| 20 | Bad Record Mac |
| 21 | Decryption Failed |
| 22 | Record Overflow |
| 30 | Decompression Failure |
| 40 | Handshake Failure |
| 41 | No Certificate |
| 42 | Bad Certificate |
| 43 | Unsupported Certificate |
| 44 | Certificate Revoked |
| 45 | Certificate Expired |
| 46 | Certificate Unknown |
| 47 | Illegal Parameter |
| 48 | Unknown CA |
| 49 | Access Denied |
| 50 | Decode Error |
| 51 | Decrypt Error |
| 60 | Export Restriction |
| 70 | Protocol Version |
| 71 | Insufficient Security |
| 80 | Internal Error |
| 90 | User Canceled |
| 100 | No Renegotiation |
| 110 | Unsupported Extension |

## A.3 Modes of Operation

**ECB** Electronic Codebook

**CBC** Cipher Block Chaining

**OFB** Output Feedback

**CFB** Cipher Feedback

**CTR** Counter

**XTS-AES** XEX Tweaked CodeBook for Advanced Encryption Standard

**CMAC** Cipher-based Message Authentication Code

**CCM** Counter with CBC-MAC

**GCM** Galois/Counter Mode

Appendix B

# Source Code

## B.1 Compile and Run the Application

To compile and run the application, start a command prompt, navigate to the source code folder, and execute the following commands:

```
// compile the source code into Java bytecode
javac Main.java
// executes the application
java Main
```

Listing B.1: Compile and Run EduTLS

Note that compiling the source code required write access to the directory, and can therefore not be done directly on the CD.

## B.2 SHA-256 Pseudocode

**Operations and Symbols**
All addition (+) in the algorithm is performed with modulo $2^{32}$ of the result, which means that $x+y$ is calculated as $(x+y) mod\ 2^{32}$.

The $\oplus$ sign is the bitwise logical *XOR* operation, the $\vee$ sign refers to *OR* and $\wedge$ refers to the logical *AND* operation.

| a | b | a XOR b | a OR b | a AND b |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |

*Figure 1 - Truth table for XOR, OR and AND*

$<<$ is bitwise right shift operation, often referred to as $ROTR^n(x)$, where $n$ bits from

the right side are discarded and $n$ '0' bits are appended to the left side. >> performs the opposite, a left shift operation referred to as *SHR*, where $n$ bits from the left side are discarded, and $n$ '0' bits are appended to the right side.

| a | a << 2 | a << 4 | a >> 2 | a >> 4 |
|---|---|---|---|---|
| 110101100001101 | 001101011000011 | 000011010110000 | 010110000110100 | 011000011010000 |

*Figure 2 - Example of ROTR and SHR*

**Preprocessing**
The first step is to initialize the 64 round constants, $K_1$ to $K_{63}$, with the first 32 bits of the fractional parts of the cube roots of the first 64 primes. As an example, prime number 64 is 311, and $\sqrt[3]{311} = 6.7751689522$, separate the fractional part gives 0.7751689522, and convert it to hex; $K_{64} = 0xc67178f2$.
Before the hash computation starts, three preprocessing steps of the message are performed. The first, message padding, is to ensure the length of each chunk is 512 bits. Begin with appending a binary '1' to the message, and then append as many '0' bits as necessary to make the total message length equal to 448 (mod 512)[1]. The last 64 bits[2] is the initial length of the message expressed binary. The second preprocessing operation is to break the message into $N$ 512 bit chunks. The last step before the hash computation is to initialize eight variables, $H_0$ to $H_7$, with the first 32 bits of the fractional parts of the square root of the first eight primes. For instance, $H_0$ will calculated by $\sqrt{2} = 1.41421356$, separate the fractional part gives 0.41421356, and convert to hex result in *0x6a09e667*.

**Hash Computation**
Each of the $N$ chunks are processed with the following steps. Create 64 variables, $W_1$ to $W_{63}$, referred to as words. First, break the 512 bit message into sixteen 32 bit chunks and assign the values to $W_1$ to $W_{15}$. For $W_{16}$ to $W_{63}$, compute the values with the formula
$W_t = \sigma_1 + W_{t-7} + \sigma_0 + W_{t-16}$
where $\sigma_0 = ((W_{t-15} >> 7) \vee (W_{t-15} << 25)) \bigoplus (W_{t-15} >> 18) \vee (W_{t-15} << 14) \bigoplus (W_{t-15} >> 3)$
and $\sigma_1 = ((W_{t-2} >> 17) \vee (W_{t-2} << 15) \bigoplus (W_{t-2} >> 19) \vee (W_{t-2} << 13) \bigoplus (W_{t-2} >> 10))$

After these steps, all the $W$ words has been assigned values. Next is to initialize the eight working variables, a,b,c,d,e,f,g and h with the $(i-1)^{st}$ hash value; $a = H_0^{i-1}$
$b = H_1^{i-1}$
$c = H_2^{i-1}$
$d = H_3^{i-1}$
$e = H_4^{i-1}$
$f = H_5^{i-1}$
$g = H_6^{i-1}$
$h = H_7^{i-1}$

The next operation is to perform a 64 step iteration to manipulate the variables a-h. In this iteration, two temporary variables, T1 and T2, are used.

Calculate

[1] Modulus, the rest from an integer division between two numbers.
[2] 512 - 448 = 64

$T1 = h + \Sigma 1 + ((e \wedge f) \bigoplus (\neg e \wedge g)) + K_t + W_t$
and
$T2 = \Sigma 0 + ((a \wedge b) \bigoplus (a \wedge c) \bigoplus (b \wedge c))$
where
$\Sigma 1 = (e >> 6) \vee (e << 26) \bigoplus (e >> 11) \vee (e << 21) \bigoplus (e >> 25) \vee (e << 7)$
and
$\Sigma 0 = (a >> 2) \vee (a << 30) \bigoplus (a >> 13) \vee (a << 19) \bigoplus (a >> 22) \vee (a << 10)$

The rest of the variables goes as followed:
h = g
g = f
f = e
e = d + T1
d = c
c = b
b = a
a = T1 + T2

When this 64 step iteration is finished, the last step is to compute the new hash values for $H^{(i)}$ with $H_0^{(i)} = a + H_0^{(i-1)}$, $H_1^{(i)} = b + H_1^{(i-1)}$ and so on. As mentioned, all these operations are performed on every $N$ 512 bit chunks. The final operation is to concatenate the $H_0$ to $H_7$ values, each of 32 bits, which results in a 256 bit message digest.

## B.3 Calculate SHA-1 and SHA256 Digest

This appendix provides the source code used to calculate the predefined digest values that is used in the unit test that verifies the hash functions.

```
1  public void calculatePredefinedDigest() {
2      String test1 = "";
3      String test2 = "abc";
4      String test3 = "abcdefghijklmnopqrstuvwxyz";
5
6      String test1sha1digest = SHA1(test1);
7      System.out.println(test1sha1digest);
8      // DA39A3EE5E6B4B0D3255BFEF95601890AFD80709
9      String test2sha1digest = SHA1(test2);
10     System.out.println(test2sha1digest);
11     // A9993E364706816ABA3E25717850C26C9CD0D89D
12     String test3sha1digest = SHA1(test3);
13     System.out.println(test3sha1digest);
14     // 32D10C7B8CF96570CA04CE37F2A19D84240D3A89
15
16     String test1sha256digest = SHA256(test1);
17     System.out.println(test1sha256digest);
18     // E3B0C44298FC1C149AFBF4C8996FB92427AE41E4649B934CA495991B7852B855
19     String test2sha256digest = SHA256(test2);
20     System.out.println(test2sha256digest);
21     // BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD
22     String test3sha256digest = SHA256(test3);
23     System.out.println(test3sha256digest);
```

```
24    // 71C480DF93D6AE2F1EFAD1447C66C9525E316218CF51FC8D9ED832F2DAF18B73
25    }
26
27 public String SHA1(String input)
28          throws NoSuchAlgorithmException, UnsupportedEncodingException  {
29     MessageDigest md = MessageDigest.getInstance("SHA-1");
30     byte[] sha1digest = new byte[40];
31     md.update(input.getBytes("UTF-8"), 0, input.length());
32     sha1digest = md.digest();
33     return toHexString(sha1digest);
34     }
35
36 public String SHA256(String input)
37          throws NoSuchAlgorithmException, UnsupportedEncodingException  {
38     MessageDigest md = MessageDigest.getInstance("SHA-256");
39     byte[] sha256digest = new byte[64];
40     md.update(input.getBytes("UTF-8"), 0, input.length());
41     sha1digest = md.digest();
42     return toHexString(sha256digest);
43     }
44
45 public static String toHexString(byte[] array) {
46    StringBuffer buffer = new StringBuffer (array.length*2);
47    for (int i=0;i<array.length;i++)
48       buffer.append (s_hexUpper[array[i] & 0xFF]);
49    return  buffer.toString();
50    }
```

Listing B.2: SHA Digest Calculation

## B.4 Generating Diffie-Hellman P and G

The listing below demonstrates how the P and G values are generated.

```
1 // Throws NoSuchAlgorithmException
2 AlgorithmParameterGenerator paramGen = AlgorithmParameterGenerator.
    getInstance("DH");
3 //Initializes a 512 bits parameter generator
4 paramGen.init(512);
5 // Generates the parameters.
6 AlgorithmParameters params = paramGen.generateParameters();
7 // Gets a (transparent) specification of the parameter object
8 DHParameterSpec dhSkipParamSpec = (DHParameterSpec)params.getParameterSpec(
    DHParameterSpec.class);
9 // Throws NoSuchAlgorithmException
10 KeyPairGenerator aliceKpairGen = KeyPairGenerator.getInstance("DH");
11 aliceKpairGen.initialize(dhSkipParamSpec);
12 KeyPair aliceKpair = aliceKpairGen.generateKeyPair();
13 KeyAgreement aliceKeyAgree = KeyAgreement.getInstance("DH");
14 aliceKeyAgree.init(aliceKpair.getPrivate());
15 System.out.println(dhSkipParamSpec.getG().toString());
16 // 70138926809884982791885540234496571704510110195718228435
17 // 25388197555796586834894509532314177249376660084062463676
18 // 55446913871196418051438471777200947267417
19 System.out.println(dhSkipParamSpec.getP().toString());
```

```
// 74686000784664261147964638241255264122031066126097902924
// 66106804957294177866776167499103011102236201812549527633
// 94065953101173284807707891134229462124767 9
```

Listing B.3: Generating Diffie-Hellman P and G

## B.5 RSA Key Generation

```
/**
 * This method prints out
 * all coprimes to a the given number
 */
public void findCoprimesTo(int number) {
   for(int i = 3; i < number; i++) {
         if(GCD(i, number) == 1)
            System.out.println(i + " is coprime to " + number);
      }
   }

/**
 * The GCD function find the greatest
 * common divisor between two numbers.
 * If GCD is 1, the numbers are
 * coprimes
 */
public int GCD(int a, int b) {
      if (b==0) return a;
      return GCD(b, a%b );
   }
```

Listing B.4: Coprime Finder

```
/**
 * Finds the modular multiplicative inverse
 * of e mod modulus
 */
public int findMMI(int e, int modulus) {
   while(true) {
      for(int d = 3; d < modulus; d++) {
         int product = d*e;
         if((product % modulus) == 1) {
            return d;
         }
      }
   modulus = modulus*2;
   }
}
```

Listing B.5: Modular Multiplicative Inverse Finder

```
public static boolean miller_rabin(BigInteger number) {
   // if testBit is zero, it's an even number, hence not a prime
   if(!number.testBit(0))
      return false;
   BigInteger a;
```

```
6     // the test is run for 20 times to obtain high confidence that it's a
          prime
7      for (int i = 0; i < Globals.NUM_OF_PRIME_TESTS; i++) {
8        // we don't want a to be zero, but between 1 and n
9        // java.util.Random rnd
10        do {
11            a = new BigInteger(number.bitLength(), rnd);
12        } while (a.equals(BigInteger.ZERO));
13        if (!miller_rabin_pass(a, number)) {
14            return false;
15          }
16       }
17     return true;
18     }
19
20 private static boolean miller_rabin_pass(BigInteger a, BigInteger n) {
21     BigInteger n_minus_one = n.subtract(BigInteger.ONE);
22     BigInteger d = n_minus_one;
23     int s = d.getLowestSetBit();
24     d = d.shiftRight(s);
25     BigInteger a_to_power = a.modPow(d, n);
26     if (a_to_power.equals(BigInteger.ONE))
27       return true;
28     for (int i = 0; i < s-1; i++) {
29         if (a_to_power.equals(n_minus_one))
30          return true;
31         a_to_power = a_to_power.multiply(a_to_power).mod(n);
32     }
33     if (a_to_power.equals(n_minus_one))
34       return true;
35     return false;
36     }
```

Listing B.6: The Miller-Rabin Primality Test

```
1 private void generateKeys(int size) {
2       // p and q are two random primes
3       BigInteger p, q;
4       // n is used as the modulus of the public and private key
5       // e is the public key, while d is the private
6       BigInteger n, m, e, d;
7       Random rnd = new Random();
8       do {
9          p = new BigInteger(size,rnd);
10      } while(!miller_rabin(p));
11
12      do {
13         q = new BigInteger(size,rnd);
14      } while(!miller_rabin(q));
15
16      // n = p*q, the modulus of the certificate
17      n = p.multiply(q);
18      // Find how many coprimes there exists of n by using the Euler's
            totient function: m = (p-1)*(q-1)
19      m = (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE))
            ;
20      // Sets e initially to 65537, which is a common value to ensure
            efficient encryption and sufficient strength
```

```
21        e = new BigInteger("65537");
22        // Must calculate e such as the greatest common divisor of e and m is
              1
23        while(m.gcd(e).intValue() > 1)
24           e = e.add(new BigInteger("2"));
25        // Calculates d to satisfy d*e=1 (mod m)
26        d = e.modInverse(m);
27        // the RSA keys are now finished
28        }
```

Listing B.7: Generating RSA Key-Pair