Bachelor thesis
Computer Science

Radboud University

Learning state machines of TLS 1.3 implementations

Author:
Jules van Thoor
s4447921

First supervisor:
Dr. ir. Joeri de Ruiter
joeri@cs.ru.nl

Second assessor:
Dr. ir. Erik Poll
e.poll@cs.ru.nl

April 18, 2018

Abstract

The TLS (Transport Layer Security) protocol is a widely used cryptographic protocol that secures communication over networks. It provides security when browsing the internet (HTTPS), sending emails (SMTP) or, for example, connecting to a VPN. TLS 1.3, its upcoming version, introduces some drastic changes with respect to the previous TLS version (TLS 1.2), including both security and speed improvements. We automatically inferred the implemented state machines from two major TLS 1.3 implementations (OpenSSL and WolfSSL) and analysed them on unexpected behaviour, using a strategy called *state machine learning*. State Machine learning is a very useful protocol-analysis technique for various reasons, the most important being that state machines are easy to analyse and, once set up, the learning process can easily be repeated for other implementations of the same protocol. In the learning process, we made use of a mapper to send messages back and forth between the learning tools and the implementations themselves. The TLS 1.3 implementations analysed showed no behaviour that could cause serious security flaws, although one of them, WolfSSL, returned peculiar output on some of the inputs sent.

Acknowledgements

First, I want to give special thanks to my supervisor, Joeri de Ruiter, who guided me in the (sometimes cumbersome) process of conducting this research and provided me the code where I based the mapper on.
Second, my appreciation for the researchers from the Ruhr University Bochum working on the TLS-Attacker framework, both for giving me access to their project and for helping me solve some frustrating errors that I encountered during the derivation of the state machines.
Last I want to thank my fellow student Max van Deurzen for helping me improve my English and pointing out most of my typos.

Contents

Chapter 1

Introduction

In a world where our daily lives are becoming more and more dependent on software and technology, the security of our data, applications and communication is a fast-growing concern. In 2010, the global internet traffic was approximately twenty thousand petabytes per month [1]. Last year, 2016, this has already increased to 96,000 petabytes. According to Cisco, global IP traffic will again increase nearly threefold over the next 5 years, reaching 3.3 zettabytes in 2021 [2]. While this huge increase in internet traffic may be a positive advance, it is quite disturbing to know that half of the web's traffic is still not encrypted, according to a report from EFF [3].

Almost all important organisations nowadays use the internet for communication and storing, transporting and retrieving data (think of governments, banks, hospitals, insurance companies or webshops). Therefore, it is necessary that our communication over the internet is secure. We don't want others to be able to spy on our communication, to modify messages we send over social media or personal information stored on Amazon and most of all, we want to know that the party we are communicating with is who they claim to be. This can all be summarized in three requirements: confidentiality, integrity and authenticity. To meet these requirements, a lot of modern websites and servers make use of a security protocol called Transport Layer Security (TLS). When visiting a website, the small green padlock on the left of the address bar is an indication that the HTTP traffic to and from this website is encrypted using TLS, resulting in HTTPS.

Like any protocol, security related or not, TLS is not perfect. It is in need of constant improvement, which is why there are already five major versions of it released over the past two decades. From SSL 2.0 in 1995 to TLS 1.2 in 2008, the protocol has seen many changes and improvements, which made

each version more secure than the one before. Since it is already more than nine years ago that the last TLS version (TLS 1.2) was released, one might wonder whether version 1.2 will be the final TLS version. Since there have been some high profile attacks against TLS 1.2 [4], it is good that this is not the case. As of April 2014, TLS 1.3 is in the making and will bring with it the most drastic changes so far. Even though it does not have an official RFC yet but only a draft[1], many TLS 1.3 implementations are already released, including some major ones like OpenSSL 1.1.1[2] and WolfSSL 3.13.0[3].

Since most of the time it is the implementation of the protocol that causes a problem (think of the disastrous Heartbleed bug in OpenSSL in 2014 [5]) rather than the protocol itself, it would prove useful to have a technique to test a protocol implementation on unexpected behaviour. This is where *state machine learning* enters the stage. In this context, when we talk about state machine learning, we mean the derivation of a finite-state automaton from a particular implementation of a protocol, here TLS 1.3. State machines can be very helpful when testing a specific implementation on flaws or bugs. First, they are a great alternative to digging through lines and lines of code to find errors in the execution of a program. Second, the same derivation technique can be applied to all implementations of a specific protocol. Finally, when comparing multiple implementations of a protocol, it is often easier to compare the derived state machines than the code of the implementations themselves.

1.1 Aim of the thesis

In this thesis, finite state machines will be derived from the server side of two very popular TLS implementations that support TLS 1.3: OpenSSL 1.1.1 and WolfSSL 3.13.0. We will not do this by hand, but use a tool, Learnlib[4], that automates this process, since it will be easy to reproduce it this way and apply it to other TLS 1.3 implementations. After the state machines are derived, we will analyse them to see if the implementations show behaviour that they should not, according to the TLS 1.3 specification and draw our conclusions from this. We will also compare the two state machines, to see if there are any differences between them.

[1] https://tools.ietf.org/html/draft-ietf-tls-tls13-23
[2] https://github.com/openssl/openssl
[3] https://github.com/wolfSSL/wolfssl
[4] https://github.com/LearnLib/learnlib

1.2 Thesis organisation

The thesis will be organised as follows. In chapter 2, we give a brief overview of the related work done on the subject of state machine learning and security protocol analysis, especially on TLS. Chapter 3 will give an overview of the TLS 1.3 protocol and the improvements compared to previous versions. We will discuss both speed improvements and security enhancements. Chapter 4 will introduce the reader to the concept of state machine learning with all its aspects, including Mealy machines, the L* algorithm and equivalence algorithms. In chapter 5 we focus on implementation specific details. We discuss the configurations used (i.e. cipher suites and key exchange methods), the input and output alphabet for the state machines and the necessary tools to derive the state machines from the implementations. We also give a brief overview of OpenSSL and WolfSSL and our final setup. Chapter 6 will be entirely devoted to the analysis of and comparison between the inferred state machines and their possible undesirable behaviour. Finally, in chapter 7 we discuss our conclusions and give suggestions for future work.

Chapter 2

Related Work

Several studies have been done on the use of state machine learning to analyse implementations of both security-related protocols (like IPSec, OpenVPN, SSH, TLS [6, 7, 8, 9]) and non security-related protocols like TCP [10]. These studies all focused on one or more implementations and derived state machines from them. Based on these machines, the implementations were analyzed on unexpected behaviour or security flaws.
Below, we discuss the studies that are most related to our own research, namely the two that also focused on TLS. We give a quick overview of the methods used and the most prominent results.

Nine implementations of TLS 1.0 and 1.2 were analysed in 2015 by De Ruiter et al. [9], including OpenSSL, Java Secure Socket Extension (JSSE) and GnuTLS. These were tested both server side and client side, using a test harness supporting client certificate authentication and the HeartBeat Extension. The state machines were inferred using LearnLib, which utilized a modified version of the L* algorithm to construct the hypotheses and an improved version of Chow's W method [11] for equivalence testing.
The result was that three of the nine analysed implementations contained previously undiscovered security flaws. One of them (Java Secure Socket Extension[1]) even contained a bug that made it possible for the server to accept plaintext data. The method used here to learn the state machines from these implementations will form the base for our own approach.

Another research in the field of TLS and state machines was done by Beurdouche et al. in 2015 [12]. Various TLS 1.0 implementations (including OpenSSL 1.0.1) were tested for state machine bugs, using FlexTLS[2]. FlexTLS

[1]https://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html
[2]https://github.com/mitls/mitls-flex

is built upon MiTLS[3] (a verified reference TLS implementation) and is a tool for prototyping and testing TLS implementations.
Serious flaws were discovered, most of them in OpenSSL. For example, sending an early `ChangeCipherSpec` message triggered derivation of a record key from the session key, which in turn could lead to client and server impersonation attacks.

In 2016, a total of 145 OpenSSL and LibreSSL versions were tested both server side and client side by Joeri de Ruiter [13], also by using state machine inference. Security vulnerabilities and erroneous behaviour was observed in many versions, together with the point in time where it was fixed by the developers [13].

[3]https://github.com/mitls

Chapter 3

TLS 1.3

In this chapter, we will accustom the reader to the TLS 1.3 protocol. We will first give a detailed description of the protocol itself (section 3.1). This includes three sub-protocols: the Handshake Protocol, the Record Protocol and the Alert Protocol. After this, we compare TLS 1.3 with version 1.2. We will discuss the differences and improvements made in TLS 1.3 regarding its predecessor in section 3.2. The changes made in TLS 1.3 can be divided into two categories: security enhancements and speed improvements.

3.1 TLS 1.3 overview

TLS 1.3 consists of three sub-protocols: the Handshake Protocol, used to establish the TLS session, the Record Protocol, which transmits, fragments, decrypts and reassembles messages and the Alert protocol, which contains messages indicating that some kind of problem occurred. Below, we will discuss each of them and describe the different types of messages they contain. Note: the content of this overview is loosely based on the information provided in the current working draft (21) of TLS 1.3[1].

3.1.1 Handshake Protocol

The handshake protocol is perhaps the most complex part of TLS 1.3. During the handshake phase, the following takes place between client and server:

- Cipher suites are negotiated;
- The server (and possibly the client) is authenticated;

[1] https://tools.ietf.org/html/draft-ietf-tls-tls13-21

- Keys are exchanged for encryption and decryption.

If all goes well, a secure session should be established between client and server at the end of this phase. Figure 3.1 shows a basic TLS 1.3 handshake. The messages in parentheses are optional.

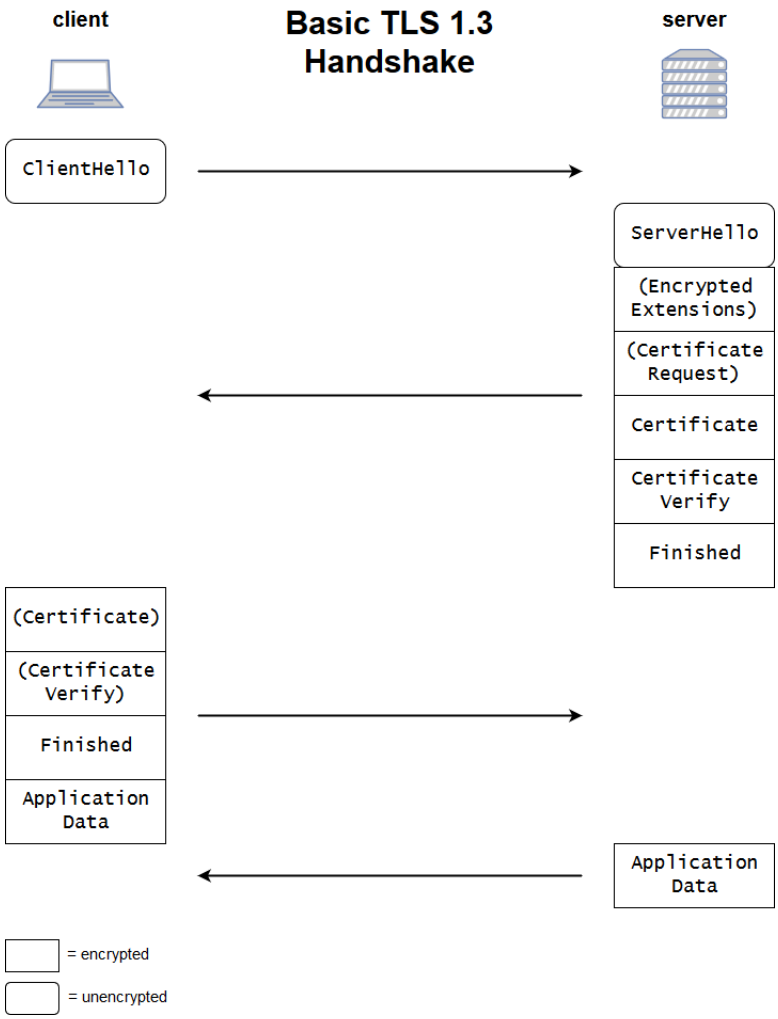

Figure 3.1: Depiction of a basic TLS 1.3 handshake, with optional messages in parentheses.

The Handshake Protocol can be divided into three phases: the Key Exchange (including the `ClientHello` and `ServerHello`), the Server Parameters (`EncryptedExtensions` and `CertificateRequest`) and the Authentication (`Certificate`, `CertificateVerify` and `Finished`). In the remaining

part of this subsection, we will explain each phase as well as the messages they contain.

Key Exchange

In this phase, the client and server agree on cipher suites and establish shared keying material for the rest of the handshake. The messages in this phase are not encrypted.

- `ClientHello`: The `ClientHello` message initiates the handshake. The most important components are: a random 32 byte integer, a list of supported cipher suites and the list of extensions. Of these extensions, there are two that are necessary for the handshake to succeed:

  - `supported_versions`: indicates the preferred TLS versions;
  - `signature_algorithms`: the signature algorithms the client accepts.

  In addition to this, there are also extensions in the `ClientHello` that are mandatory for the key exchange. In TLS 1.3 there are two options for the key exchange mode: (Elliptic Curve) Diffie-Hellman Ephemeral ((EC)DHE) or Pre Shared Key (PSK). If (EC)DHE is desired, the following two extensions need to be included:

  - `supported_groups`: a list of supported (EC)DHE groups supported by the client;
  - `key_share`: the (EC)DHE key shares for each of the preceding groups.

  For PSK mode, the mandatory extensions are:

  - `pre_shared_key`: a set of PSK labels to be used for the handshake;
  - `psk_key_exchange_modes`: the mode to be used with PSK. Either PSK-only or PSK with (EC)DHE key establishment.

- `ServerHello`: The server responds to the `ClientHello` with a `ServerHello`. The `ServerHello` consists of: the TLS version used, a random 32 byte integer, the cipher suite selected by the server from the clients extension-list and a list of extensions. The only extensions that are allowed in the `ServerHello` are the `key_share` extension (in case of (EC)DHE) or the `pre_shared_key` extension (in case of PSK).

After the `ClientHello` and `ServerHello`, the HMAC-based Key Derivation Function (HKDF)[2] as described in RFC 5869[3] is used to create the keys for encrypting and decrypting the messages in the rest of the handshake. The resulting keys are called the `client_handshake_traffic_secret` and `server_handshake_traffic_secret`.

Server Parameters

The messages contained in this phase are both optional. Among other things, it is decided here if client authentication is needed.

- `EncryptedExtensions`: contains extensions that are not necessary for the cryptographic parameters, i.e., `server_name`, `supported_groups` and `client_certificate_type`.
- `CertificateRequest`: indicates if client authentication is desired.

Authentication

The authentication phase concludes the handshake. Server (and client) are being authenticated, keys are confirmed and the integrity of the handshake is verified.

- `Certificate`: the certificate used for server (and possibly client) authentication.
- `CertificateVerify`: the `CertificateVerify` message serves two purposes. First, it proves to the other endpoint that its certificate is correct (by showing that it possesses the corresponding private key). Second, it provides integrity for all the handshake messages used so far. This is accomplished by the so-called `Transcript-Hash`. The `Transcript-Hash` is a hash computed over the concatenated content (including headers) of all the handshake messages received so far. For the server, this is a hash over the `ClientHello`, `ServerHello`, `EncryptedExtensions` and `Certificate`.
- `Finished`: the final message of the authentication phase, `Finished`, is sent by both the client and server. An HMAC is computed over the following two components:

[2]HKDF is used in TLS 1.3 to convert shared secrets into useful keying material.
[3]https://tools.ietf.org/html/rfc5869

- `finished_key`: the output of the HKDF function described earlier over the `client/server_handshake_traffic_secret`, the string "finished" and the length of the hash algorithms output
- `Transcript-Hash`: the same `Transcript-Hash` as in the `CertificateVerify`, only now including all the additional messages received so far

The `Finished` message of one endpoint (client or server) must be verified by the other, before any application data can be sent. If one endpoint finds out that the other endpoints `Finished` is incorrect, it should terminate the connection with an appropriate alert.

3.1.2 Record Protocol

The Record Protocol handles the communication between client and server and is, therefore, a very important component of TLS 1.3. On the sending side, each message is processed by the Record Protocol as follows:

1. The message is chopped up into fragments.
2. (a) If encryption is not needed yet, the record is sent as a so-called `TLSPlaintext` message, containing the type of the record (either handshake, application data or alert), the length and the record itself.

   (b) If encryption is needed, The record is first converted to a `TLSInnerPlaintext` message, containing the record itself, the type and an amount of zeroes for padding. Next, this `TLSInnerPlaintext` is encrypted by the AEAD (Authenticated Encryption with Associated Data) algorithm, using the client/server write-key and a nonce[4]. The output is placed inside a `TLSCiphertext` message, together with its length and a few other fields.
3. The message, being either `TLSPlaintext` or `TLSCiphertext` is transmitted.

On the receiving side, the Record Protocol proceeds as follows:

1. If it contains `TLSCiphertext`, the received data is decrypted and verified using the same AEAD algorithm, the key and the nonce as mentioned before.

[4]The nonce represents a sequence number that is maintained by the client and server to prevent replay-attacks

2. (a) If the decryption failed, an alert message is sent and the connection is closed.
   (b) If the decryption succeeded (or there is no encryption used at all), the records are reassembled and delivered to the higher-level entity.

In figure 3.2, the different types of records are depicted.

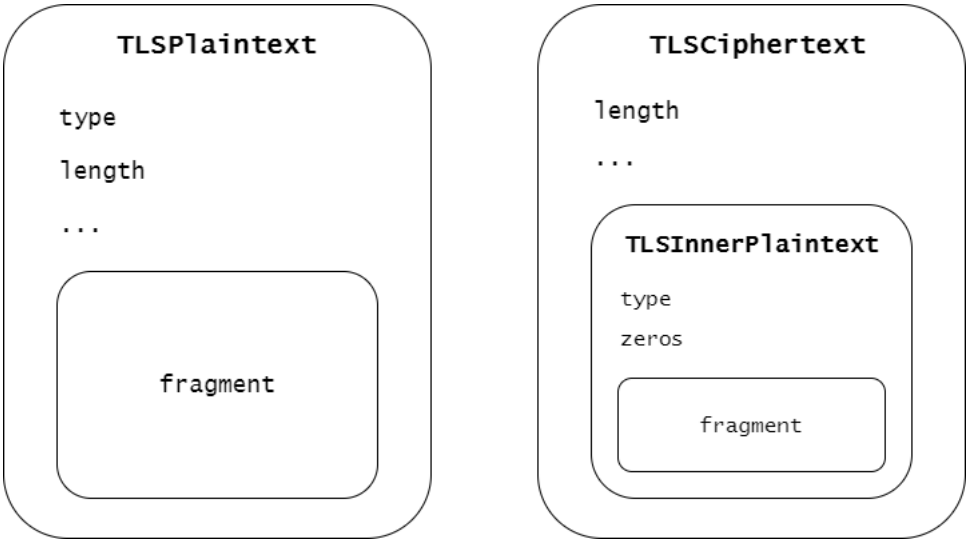

Figure 3.2: Depiction of the two types of messages in the Record Protocol: **TLSPlaintext** and **TLSCiphertext**

3.1.3 Alert Protocol

Like the name suggests, the Alert Protocol contains messages that indicate the occurrence of some kind of problem during a TLS session. Upon receiving an alert message, the receiving side should notify this to the application layer and close the connection as soon as possible. Any secret or key associated with this failed session must be forgotten. An alert consists of an alert description and an alert level. In TLS version 1.3, the level can safely be ignored since the connection is always closed upon receiving any kind of alert. Below, we will summarize the most important alert descriptions and their meanings:

- `close_notify`: an indication that the connection will be closed by the sender. Data received after this alert can be ignored;

- `decode_error`: the message does not follow the syntax specified by the TLS 1.3 protocol;
- `illegal_parameter`: the message follows the correct syntax, but contains some semantic incorrect value(s);
- `decrypt_error`: an indication that a cryptographic operation during the handshake failed;
- `insufficient_security`: the parameters offered by the client during the handshake were found insufficient by the server;
- `unexpected_message`: an indication that a message is encountered that is not appropriate at this stage;
- `unsupported_extension`: the message contained an extension that was either forbidden in the particular message or was not first offered by the client;
- `bad_record_mac`: a failure in the decryption of a certain message.

3.2 Differences from TLS 1.2

Nine years have passed since the release of TLS 1.2 in 2008. This is a relatively long time in the cybersecurity world, since technology is advancing rapidly. On top of that, attacker's skill sets improve as time progresses, resulting in some high profile attacks against TLS 1.2. In the remaining part of this chapter, we will describe the security enhancements and speed improvements made in TLS 1.3 compared to TLS 1.2.

3.2.1 TLS 1.3 security enhancements

The security enhancements made in TLS 1.3 primarily consist of the removal of TLS 1.2 features that are considered insecure:

- CBC mode ciphers
- RSA key transport
- RC4 and SHA-1 algorithms
- Arbitrary Diffie Hellman groups

Most of these features caused some high-profile attacks against TLS 1.2 (i.e. Lucky 13, BEAST and FREAK [14, 4, 15]). Therefore, in TLS 1.3 they are not used anymore.

3.2.2 TLS 1.3 Speed improvements

TLS 1.3 has made a huge advance when it comes to speed. When requesting a web page over HTTPS using TLS 1.2, it took three round trips before the actual HTML data was received (two for the handshake and one for the GET request and response).

In TLS 1.3 however, the first GET is sent together with the final handshake message (the `Finished` from the client). The combined handshake and GET now take only two round-trips, which makes it much faster than its predecessor. Figure 3.3 nicely illustrates this difference.

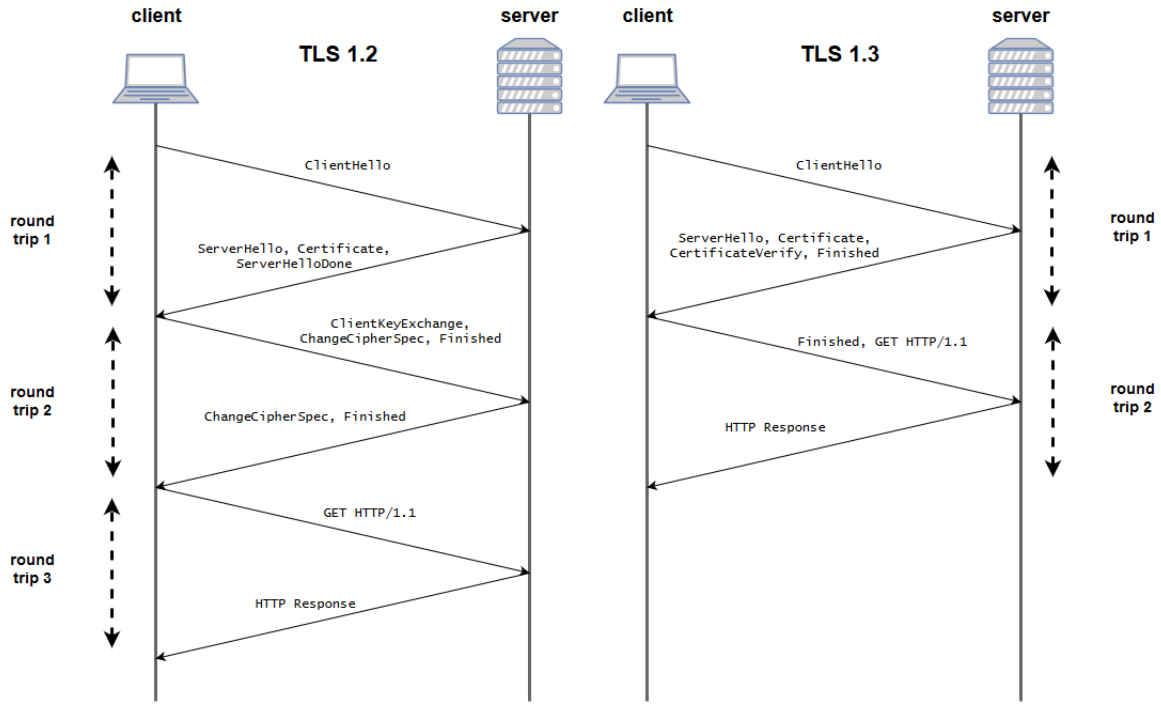

Figure 3.3: The round-trips needed for TLS 1.2 (left) and 1.3 (right)

Chapter 4

State Machine Learning

In this chapter, we will familiarize the reader with the concept of state machine learning. First, state machines are discussed in section 4.1. We will describe the type of state machines that we will use in our own research, the Mealy Machines. Section 4.2 is about the first part of the learning process, the construction of a hypothetical state machine with the L* algorithm. We also describe the equivalence algorithm *Randomwords*.

4.1 State machines

A state machine (often called a finite state machine or finite state automaton) is a mathematical model used to represent an execution flow [16]. A state machine consists of a finite set of states (Q) and transitions between them. These transitions go from one state to another, representing some kind of input that is received when in a particular state. It is only possible for the machine to be in exactly one state at a particular time. The transitions in a state machine can be described by a transition function δ:

$$\delta : (Q \times \Sigma) \rightarrow Q$$

where Σ is the *input alphabet* (a set of different symbols that represent the input at a given state). The *initial state* is called q_0 and the set of *final states* (the states in which an execution flow is 'accepted') F. With these five items (the set of states, the transition function, the input alphabet, the initial state and the set of final states) we can define any state machine as a five-tuple $(Q, \delta, \Sigma, q_0, F)$. We call this kind of state machine a *deterministic* state machine because in each state it has exactly one transition for each input symbol. In a *non-deterministic* state machine, the machine can have multiple transitions with the same input symbol from a given state, or

even have no transitions at all. In this thesis, however, we only make use of deterministic finite state machines, since we assume the implementations we analyse don't have non-deterministic behaviour.

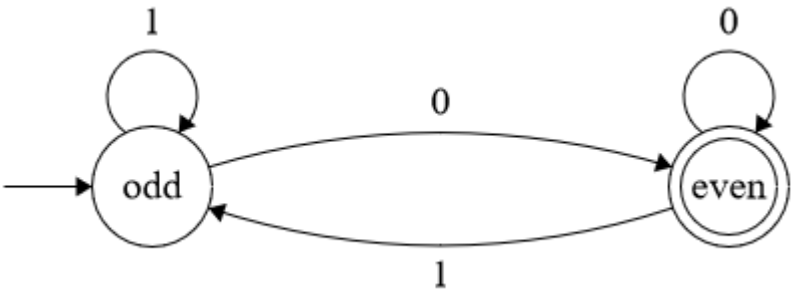

Figure 4.1: Example of a finite state machine

State machines (being either deterministic or not) can be used to model both real-world devices and computer programs. Figure 4.1 shows a simple state machine that determines for a given binary number if it is even. In this example, $Q = \{\text{odd, even}\}, \Sigma = \{0, 1\}, q_0 = \text{odd}, F = \{\text{even}\}$ and δ can be described as the function that maps each combination of state (odd or even) and input (0 or 1) to another state.

4.1.1 Mealy machines

Since this thesis is about deriving a state machine from TLS 1.3 implementations, we want the state machine to represent the general execution of such implementations as accurately as possible. The TLS handshake (and the rest of the protocol) consists of sending and receiving certain types of messages. Therefore, we need the inferred state machine to also return an output to each input message that it receives in a particular state. Fortunately, there is a type of state machine that does exactly this: the *Mealy machine*.

Mealy machines are a special kind of state machine. In addition to the state machines as we described them above, Mealy machines also return an output upon each input. This adds two extra elements to the five-tuple we used to describe general state machines: an output alphabet O (containing the various output symbols that are returned on each transition) and an output function X that computes for each combination of state and input symbol the corresponding output symbol. The output symbol is usually written next

to the input symbol on each transition, separated with a forward slash, as can be seen in figure 4.2.

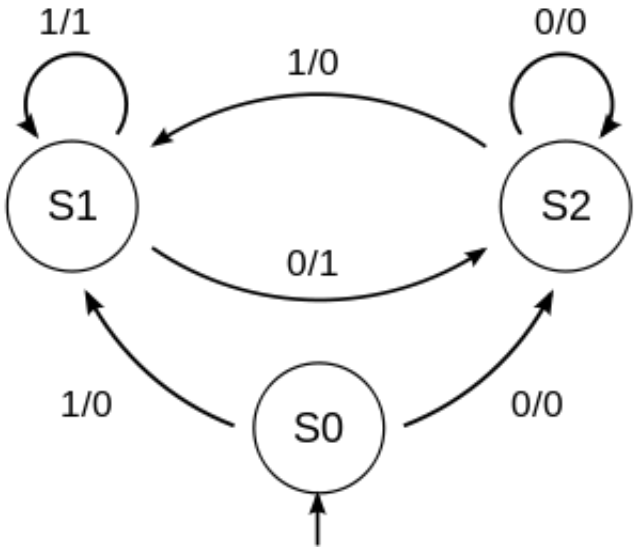

Figure 4.2: Example of a Mealy machine [27]

4.2 State Machine Learning

In this section, we will describe the process of learning the state machine. Our approach is based on the L* algorithm, which we will use to infer state machines from the TLS 1.3 implementations. The L* algorithm was designed by Dana Angluin in 1987 [17]. Its goal is to learn a finite state machine from a certain kind of system.

4.2.1 General overview

The learning process consists of two parts: the *learner* and the *teacher*. The learner tries to infer the state machine, by communicating with the teacher. Normally, the teacher knows the state machine and answers the learner's questions. In our case, however, the state machine is not known by the teacher, so we will use the TLS implementation itself as the teacher. Below we will sketch the four operations that take place when we infer state machines from TLS 1.3 implementations:

1. The learner tries to build a hypothetical state machine by continuously sending probe messages to the teacher (in this case the TLS 1.3 implementation itself).

2. The teacher answers the learner's messages with TLS 1.3 response messages (or no message at all).

3. Once the learner has formed its hypothetical state machine, it asks the teacher to check if this machine matches the actual state machine.

4. Since the teacher does not know the state machine in our case, it uses an equivalence algorithm to check the client's hypothesis (More about this below).

   (a) If the algorithm returns true, the teacher will let the learner know that the state machine is assumed to be correct.

   (b) If not, the teacher sends a counterexample to the learner, which in turn adjusts its guess. The process now starts again.

4.2.2 Equivalence Algorithm

Below, we will give a brief description of the equivalence algorithm we used, Randomwords.

Randomwords
The Randomwords equivalence algorithm is perhaps the easiest of the two. It just sends random input sequences of a specified minimum and maximum length to the teacher. The teacher sends output back, and if this output matches the output of the hypothetical state machine, a next input sequence is sent, and so on. After the specified number of successful input sequences, the teacher concludes that the state machine is probably correct. If the outputs do not match, the teacher gives the violating input sequence back and the learner knows that the state machine is not correct. The advantage of this approach is that it can be relatively fast to execute. On the other hand, with Randomwords there is no guarantee of the correctness of the inferred state machine.

Chapter 5

Implementation

This chapter will be devoted to our implementation and setup. We will first discuss the tools needed to provide the state machines: LearnLib and StateLearner. In section 2, we present the mapper that is used to translate StateLearner's input symbols to TLS 1.3 messages and vice versa. Then, we will list the conventions used, like the cipher suites, signature algorithms and the key exchange method. The next section will be about the two TLS 1.3 implementations we considered (OpenSSL 1.1.1 and WolfSSL 3.13.0), and finally we present our complete setup.

5.1 Inferring the state machines

In this section, we will discuss the two tools needed to learn the state machines from our TLS 1.3 implementations. The first is LearnLib, an open source framework for automata learning. The second, StateLearner, is a wrapper around LearnLib, which provides us an easier way to communicate with the mapper we designed.

5.1.1 LearnLib

LearnLib, designed by the Chair for Programming Systems at the TU Dortmund University, is an open-source Java framework for active automata learning [29]. LearnLib contains both learning algorithms (i.e. the L* algorithm) and equivalence algorithms to check the hypothetical machines. We will not directly use Learnlib for our method, but instead use StateLearner, a wrapper around LearnLib.

5.1.2 StateLearner

StateLearner, designed by Joeri de Ruiter, is a tool that serves as a wrapper around the LearnLib API and makes communicating with the mapper much easier. Settings like the input alphabet, the learning and equivalence algorithms and the hostname and port need to be specified in a configuration file, after which the program uses a black-box approach to infer the state machine. For this, it sends the symbols from the input alphabet to the mapper and receives output responses back, upon which it builds the hypotheses.

5.2 Mapper

StateLearner is not designed to communicate directly with our TLS implementations. On one hand, this makes it a very versatile tool, since it can be used for implementations of many different protocols. On the other, it means that there is still some work to do before we can test OpenSSL and WolfSSL. For this, we designed a so-called mapper. The mapper is a program, written in Java, which receives the input strings from StateLearner and translates them to the corresponding TLS 1.3 messages, which will then be sent to the implementations servers. It also needs to intercept the TLS responses from the servers, convert them to strings and forward them to StateLearner.

5.2.1 TLS-Attacker

Our mapper uses a framework called TLS-Attacker[1], developed by the Chair of Network and Data Security of the Ruhr University Bochum. Amongst other things, it contains functionality to communicate with both TLS servers and clients.

5.2.2 Changes made

The mapper as we use it in our setup is based on code that is written by Joeri the Ruiter to derive state machines from TLS 1.2 implementations. The code consists of two Java files, `TLSAttackerConnector.java` and `ConnectorTransportHandler.java`. Both Java files rely on the previously mentioned TLS-Attacker framework for creating and parsing valid TLS messages.

[1] https://github.com/RUB-NDS/TLS-Attacker

The code written by Joeri de Ruiter serves as a mapper between State-Learner and TLS 1.2. Since we need to communicate with TLS 1.3, we made some changes to it for this purpose. The `ConnectorTransportHandler` could be used for TLS 1.3 right away. The most important changes to `TLSAttackerConnecter.java` are:

- Variables like `timeout`, `tlsContext` and the port to listen for State-Learner messages are declared before the constructor, instead of being given as arguments;
- We added a `setupConfig` function, which contains all TLS 1.3 specific configurations we will discuss in section 5.3;
- Functionality for the `ChangeCipherSpec` in the `sendMessage` method was removed;
- Some functionality was moved to an `initializeSession` function;
- The `processInput` method was edited to only contain valid TLS 1.3 messages ;
- Various small fixes of methods that were not up-to-date anymore;
- The class was renamed to `Mapper`.

The two files are made publicly available on `https://gitlab.science.ru.nl/vthoor/bachelorscriptie-2018/tree/master/Code`.

5.2.3 Input and output alphabet

The standard input and output alphabet for the mapper are as follows:

Input:

- `ClientHello`
- `Finished`
- `ApplicationData`

Output:

- `ServerHello`
- `EncryptedExtensions`

- `Certificate`
- `CertificateVerify`
- `Finished`
- `NewSessionTicket`
- Numerous alert messages, i.e. `UNEXPECTED_MESSAGE`, `BAD_RECORD_MAC` and `DECRYPTION_FAILED`
- `ConnectionClosed`

In addition to this, we also decided to infer state machines with an extended set of input symbols, including server side messages (`ServerHello`, `CertificateRequest`, `CertificateVerify`, `HelloRetryRequest`) and messages that would typically be used in PSK mode (`EndOfEarlyData`, `NewSessionTicket`).

5.3 Configurations used

In this part, we will discuss all the TLS specific configurations we used in our setup. We will take a look at cipher suites, signature and hash algorithms and the key share method with its supported groups.

5.3.1 Cipher suites

Cipher suites in TLS 1.3 consist of a pair of AEAD algorithm and a hash algorithm to be used with HKDF. We chose to support the following cipher suites for the handshake[2]:

- `TLS_AES_128_GCM_SHA256`
- `TLS_AES_256_GCM_SHA384`
- `TLS_CHACHA20_POLY1305_SHA256`
- `TLS_AES_128_CCM_SHA256`
- `TLS_AES_128_CCM_8_SHA256`

[2]The format in which the cipher suites are listed is the standard from the RFC draft, with the "TLS" string first, then the AEAD algorithm and at the end the hash algorithm used for HKDF, separated by underscores.

The cipher suites listed here were the only ones to be considered secure enough to remain in the TLS protocol, so we just decided to add them all to our `ClientHello` message. We added them in the exact order as listed above.

5.3.2 Signature algorithms

Signature algorithms are used during the handshake whenever a digital signature is required. An example is the certificate which has to be signed for authentication purposes. The list of TLS 1.3 supported signature algorithms is fairly large, so we just made a selection with the most common algorithm of each type that contains SHA-2 algorithms:

- `rsa_pkcs1_sha256`
- `ecdsa_secp256r1_sha256`
- `rsa_pss_sha256`

5.3.3 Key share and Groups

We use Elliptic Curve Diffie Hellmann Ephemeral (ECDHE) as key exchange method, since ECDHE is more efficient in terms of computation power than Diffie Hellmann over 'normal' groups. The curve we chose to use for this is Curve25519, one of the fastest curves available [18] with 128 bits of security.

5.4 OpenSSL 1.1.1

The first TLS 1.3 implementation we will derive state machines from is OpenSSL 1.1.1. OpenSSL is perhaps the most well-known of all the TLS implementations currently available. The majority of all web servers use OpenSSL to secure their transport layer communication. OpenSSL is written in C and completely open-source. Besides containing implementations of all the SSL and TLS protocols, it also serves as a general-purpose cryptography library.

From version 1.1.1 on, OpenSSL also includes TLS 1.3 support up to the latest RFC draft at the moment of writing (23). When learning the state machine, we will communicate with a server demo that comes shipped with OpenSSL. After creating a public key in `key.pem` and a certificate in `cert.pem`,

we can run the server with the command:

```
$ openssl s_server -cert cert.pem -key key.pem -p 4433 -HTTP
```

5.5 WolfSSL 3.13.0

The other implementation we test, WolfSSL 3.13.0, is a more lightweight TLS library. It is open-source and also written in C. In terms of size, WolfSSL is (according to the developers[3]) about 20 times smaller than OpenSSL, and its focus lies primarily on Internet of Things (IoT).

WolfSSL includes TLS 1.3 (draft 22) support from version 3.11.1 on and comes, just like OpenSSL with a client and server demo. We use this server-demo to derive our state machine, with the command:

```
$ .wolfss/examples/server/server -v 4 -p 4433 -c cert.pem -k
key.pem -d -x -i
```

As with OpenSSL, we let the server run on port 4433 (because our mapper is configured to connect to that port). The `v` argument specifies the TLS version (1.3), the `c` and `k` arguments provide the certificate and public key files and the `d`, `x` and `i` arguments disable client authentication and let the server loop indefinitely.

5.6 Complete setup

Our complete setup is as in Figure 5.1 below. StateLearner, relying on LearnLib, sends its input symbols as strings to our implemented mapper. The mapper uses the TLS-Attacker framework (as described above) to translate these strings to real TLS 1.3 messages, which are sent as bytes to OpenSSL and WolfSSL. The implementations may respond with other TLS 1.3 messages (or alerts), which are returned as bytes to the mapper. The mapper translates this again to string format and forwards it to StateLearner. StateLearner now uses this response in the learning process, and after repeating this cycle a certain amount of times is able to give us the state machine corresponding to the TLS 1.3 implementation.

[3]https://www.wolfssl.com/docs/wolfssl-openssl/

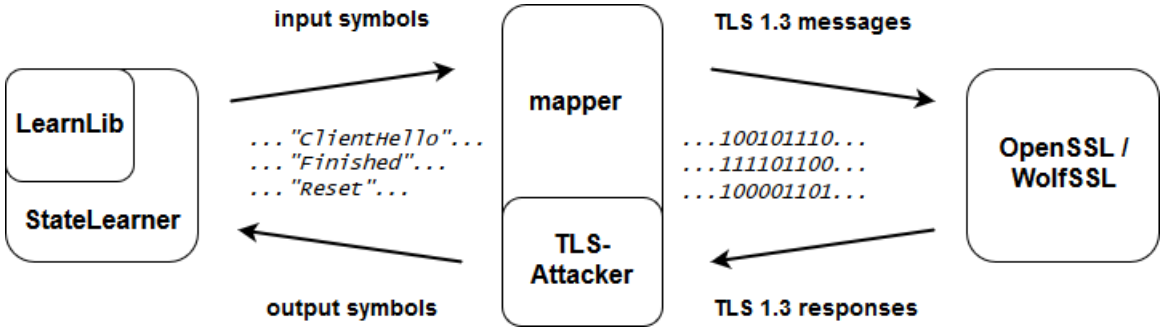

Figure 5.1: Complete overview of all the tools needed

Chapter 6

Analysis

In this chapter, we will analyse the state machines we derived from the tested TLS 1.3 implementations. We begin with inspecting OpenSSL's server side state machines and then do the same for WolfSSL. After that, we will compare the two to see if they show different behaviour when sending the same messages. We tried to learn the state machines multiple times to avoid inconsistencies.

6.1 The OpenSSL state machines

We will now evaluate OpenSSL's behaviour when sending both normal messages and messages that do not belong to the standard execution flow. We configured OpenSSL with the `enable-tls1_3` option to enable TLS 1.3.

6.1.1 Standard messages

In Appendix A.1, the complete OpenSSL 1.1.1 server side state machine can be found. In this section, we will discuss the states and transitions in the state machine and which kind of messages cause certain reactions from the OpenSSL server. When a message produces an alert, we also explain what the alert means in this situation, and what (probably) caused it.

The first state machine to discuss is the OpenSSL 1.1.1 server side state machine with only the standard client input symbols. For each state here, we will describe the outgoing transitions with their input and output values.

State 0: This is the state we enter once we have set up the connection. Normally, the first message during the TLS 1.3 handshake phase should be

a ClientHello. When sending the ClientHello, we see that the OpenSSL server responds as expected, giving back ServerHello, EncryptedExtensions, Certificate, CertificateVerify and Finished messages. Sending either a Finished or an ApplicationData at this stage results in UNEXPECTED_MESSAGE alerts, after which the connection is closed. These alerts are exactly what we expected, since (according to the specification) a ClientHello must always be sent by the client before sending or receiving any other messages.

State 1: The state machine enters this state once a correct ClientHello message has been received in the initial state. When sending another ClientHello at this point, we again get an UNEXPECTED_MESSAGE alert, which is what one should expect. Sending ApplicationData results in another alert, the DECRYPTION_FAILED alert. This alert is caused by the fact that no Finished has been sent, which means that the encryption keys are not yet authenticated. As can be seen in the state diagram in Appendix A.1, the Finished transition leads us to State 3, while receiving as output a NewSessionTicket message. This message includes a so-called *Pre Shared Key* (PSK) to resume the handshake in the future.

State 2: This state is the least interesting, since entering this state means that the connection is closed, and all further messages will get no response.

State 3: State 3 can be reached within the normal execution flow, once the Finished is sent. Since the server already received a Finished, it now expects to receive ApplicationData messages. Doing so results in ApplicationData, but sending either another ClientHello or a Finished at this point returns respectively an UNEXPECTED_MESSAGE and a BAD_RECORD_MAC alert. The latter seems a bit strange, because an UNEXPECTED_MESSAGE alert would be more fitting at this point. When we look up the description of the BAD_RECORD_MAC in the RFC draft, it says:

> This alert is returned if a record is received which cannot be deprotected. Because AEAD algorithms combine decryption and verification, and also to avoid side channel attacks, this alert is used for all deprotection failures.

Deprotection in this context stands for both decryption and verification of the received message. Although the exact cause of the alert is left vague on purpose by the designers (as can be read in the description above), the cause for this alert probably lies in the fact that the HMAC in the latter Finished

is now also computed over the content of the earlier `Finished` (which is not correct), leading to the verification at the server side not succeeding. The OpenSSL debug output offered no clues on this.

6.1.2 Adding extra input symbols

After the previous state machine (see A.1) was established, we decided to also include server input symbols to increase the chance of detecting unexpected behaviour in OpenSSL. The complete state machine can be found in Appendix A.2.

We chose to add the following input symbols to StateLearners alphabet:

- `ServerHello`
- `EncryptedExtensions`
- `CertificateVerify`
- `CertificateRequest`

The addition of the server input symbols resulted in a state machine with the same number of states but with some extra transitions. We see that all of the server input symbols result in `UNEXPECTED_MESSAGE` alerts, except when sending a `ServerHello` after a `ClientHello`, which gives `ApplicationData` in return. When examining the OpenSSL debug info however, we see that this data is not actual `ApplicationData`, but an `UNEXPECTED_MESSAGE` alert that is not correctly decrypted[1] by the TLS-Attacker framework. According to the developers, the problem is caused by the fact that TLS 1.3 does not contain a `ChangeCipherSuite` message, which leads to the framework not being able to correctly determine if the returned message is decrypted or not at this point. It is very likely that this problem is already fixed by the developers at the time of reading.

6.2 The WolfSSL state machines

We will now use the same approach to examine WolfSSL. Just like before, we begin with sending standard input symbols and extend this with server input symbols. The complete diagrams can be found in Appendix A.3 and

[1]According to the RFC draft, alerts should be encrypted once the keys have been established, e.g. after the `ClientHello and ServerHello`

A.4. When configuring WolfSSL, we used parameters `--enable-tls1_3` and `--enable-curve25519` to respectively enable TLS 1.3 and Curve25519 for the key exchange.

6.2.1 Standard messages

State 0: from the initial state, there are, just like the OpenSSL state machine, three transitions. The first one, representing a `ClientHello`, results in the standard `ServerHello`, `EncryptedExtensions`, `Certificate`, `CertificateVerify` and `Finished` messages and takes us to state 1. The other two (`Finished and ApplicationData`) result in an `UNEXPECTED_MESSAGE` alert with the connection being closed.

State 1: After the "correct" `ClientHello` has been sent, sending a `Finished` takes us to state 4. There is no `NewSessionTicket`, since WolfSSL does not have this implemented yet. Another `ClientHello` results in a `BAD_RECORD_MAC` alert and a closed connection. This alert could be considered fairly strange at this point, since the `handshake_traffic_secrets` are already established, so WolfSSL should at least be able to decrypt the "out-of-order" `ClientHello`. However, this alert is probably sent because the WolfSSL server expects encrypted messages after the `ClientHello` (which is, together with the `ServerHello` the only message that should be sent over plaintext) and not another unencrypted `ClientHello`. Sending `ApplicationData` closes the connection and returns a `WARNING_CLOSE _NOTIFY` alert.

State 2: This state is comparable with state 2 in the OpenSSL state machine. The connection is closed here, so the WolfSSL server does not return anything.

State 3: Since the `Finished` has already been sent, `ApplicationData` results (as expected) in `ApplicationData` from the server. Remarkable is that `ClientHello` and `Finished` messages at this point give back a whole list of alerts (`BAD_RECORD_MAC`, `UNEXPECTED_MESSAGE`, `UNDEFINED_UNKNOWN`, `UNDEFINED_CERTIFICATE_BAD_HASH_VALUE`). This will probably be a bug in the WolfSSL server, because when we consult WolfSSL's debugging info we indeed see multiple errors upon receiving the messages (which means it is not just data that is wrongly interpreted by the TLS-Attacker framework). We reported this issue to the WolfSSL developers. After this response, the server closes the connection.

6.2.2 Adding extra input symbols

When extending the standard input alphabet with `EncryptedExtensions`, `Certificate`, `CertificateRequest`, `CertificateVerify`, `HelloRetryRequest`, `EndOfEarlyData` and `NewSessionTicket` messages, the resulting state machine is (apart from its increased size) quite different from the one we derived before:

- All the extra transitions from the initial state cause the same type of alert (`UNEXPECTED_MESSAGE`)
- After the first `ClientHello` has been sent, all additional messages except `CertificateVerify` and `NewSessionTicket` cause a `WARNING_CLOSE_NOTIFY` alert, while the latter result in a `BAD_RECORD_MAC` alert. Of these two `BAD_RECORD_MAC` alerts, the first can be explained by the fact that no `Certificate` is sent, so the `Transcript Hash` in the `CertificateVerify` is not computed over a `Certificate` message, which makes the verification fail. The other, being the result of sending a `NewSessionTicket` after a `ClientHello`, could be caused by the server not being able to handle NewSessionTickets (which are encrypted) correctly. Still, all messages except `Finished` cause the connection to be closed immediately.
- When the correct `Finished` is sent to the server, sending a `NewSessionTicket` gives us no output, but leads us to an extra state. `CertificateVerify` and `EndOfEarlyData` seem to result in `ApplicationData`, but when examining this in the WolfSSL debugging info, we just see two alerts with descriptions *out of order error* and *sanity check on message order error*. This means that the output is just wrongly interpreted by the TLS-Attacker framework. All the other input symbols give us the same list of alerts as in the original WolfSSL state machine. All messages except the `NewSessionTicket` and `ApplicationData` cause the connection to be closed immediately.
- From the new state, all transitions lead to the "dump" state that represents a closed connection, and return the list of alerts that we described in the previous subsection, except `CertificateVerify`, `EndOfEarlyData` and a new `NewSessionTicket`. These all return `ApplicationData`, before the connection is closed. However, when examining this in WolfSSL's debugging info, we see that this is not real application data, but alerts that are not handled properly by TLS-Attacker. The error descriptions WolfSSL assigns to these messages are respectively: *out of*

order error, *sanity check of message order error* and *duplicate handshake messages*.

6.3 A comparison

We will now compare the state machines derived from OpenSSL and WolfSSL by listing the major differences between them:

- The second state machine of the WolfSSL server has one additional state, which can be reached when sending a `NewSessionTicket` after a `ClientHello` and a `Finished`. The WolfSSL server gives no response to this message, instead of closing the connection immediately like OpenSSL.
- After sending a `ClientHello` and `Finished`, incorrect messages to the OpenSSL server result in either an `UNEXPECTED_MESSAGE` or a `BAD_RECORD_MAC` alert. WolfSSL gives back a whole list of alerts.
- WolfSSL does not send a `NewSessionTicket` when the client sends its `Finished`, whereas OpenSSL does return this ticket.
- A `CertificateVerify` or `EndOfEarlyData` after the clients `Finished` make WolfSSL first return `ApplicationData`, while OpenSSL just sends alerts and closes the connection immediately.
- On numerous occasions (see the previous sections) the two implementations return different kinds of alerts.

Chapter 7

Conclusions

In this chapter, we will present the conclusions of our research, after which we will suggest further work that could be done on the subject of state machines and TLS 1.3.

7.1 Conclusions

We will now give our conclusion about each part of the research done in this thesis:

OpenSSL

The server side of OpenSSL 1.1.1 contained no unexpected behaviour that we could track using state machine learning. Whenever we sent a message that 'should not be there', we got an alert back from the server and the connection was closed. The type of alert sometimes was not what one should expect (see section 6.1), but overall the connection was always closed, which means that none of the 'wrong' messages could cause in itself any harm. Of course it is not guaranteed that if some messages are, e.g., sent a thousand times to the server this would still be the case, but that falls outside of the scope of this research. So, we can conclude that TLS 1.3 and OpenSSL made some serious improvements to respectively the TLS protocol and the old OpenSSL versions (regarding the bugs that were found in past research [9, 12]).

WolfSSL

The results regarding WolfSSL's two state machines where almost the same. We found no behaviour in the state machine that could cause serious damage, since almost every out-of-order message caused the connection to be closed.

The amount and type of alerts were sometimes not what one would expect, but except confusion at the user's side, we see no harm there. We can, therefore, conclude that WolfSSL's TLS 1.3 implementation does not contain any security flaws that could be detected by state-machine learning.

7.2 Future work

There are a lot of ways to expand the research done in this thesis. We roughly divide them into three categories:

Different methods of analysis

Instead of using state machines to analyse TLS 1.3 implementations, a different method can be used, for example:

- **Fuzzing:** a technique to test software or protocol implementations by sending large amounts of random data with the goal to trigger unexpected behaviour at the client- or server side [19]. Software tools can then be used to examine the cause of this crash.
- **Symbolic execution:** a way of analyzing which inputs cause which execution paths in an implementation. With symbolic execution, one can find bugs or infeasible paths in a programs execution.

Another protocol implementation

One could also use state machine learning to analyse different security protocols. The protocols described in the Related Work section are already examined in the past, but for i.e. Fast Initial Link Setup[1] (FILS) there is still some work to do.

Other areas of TLS 1.3

The research we did into TLS 1.3 using state machine derivation can also be expanded by examining more TLS 1.3 implementations, like the ones that can be found here:

`https://github.com/tlswg/tls13-spec/wiki/Implementations`

It is also possible to add more extensions to the mapper, for example Heartbeat, Early Data, Cookie or to use pre-shared keys instead of ECDHE.

[1] http://www.isaet.org/images/extraimages/P1215040.pdf

Bibliography

[1] Cisco, *Cisco Visual Networking Index: Forecast and Methodology, 2010–2015*, June 11, 2011.

[2] Cisco, *Cisco Visual Networking Index: Forecast and Methodology, 2016–2021*, September 15, 2017.

[3] EFF, *EFF: Half of web traffic is now encrypted*, February 22, 2017. `https://techcrunch.com/2017/02/22/eff-half-the-web-is-now-encrypted/`

[4] Rizzo, J. and Duong, T., *Browser Exploit Against SSL/TLS*, 2011. `http://packetstormsecurity.com/files/105499/Browser-Exploit-Against-SSL-TLS.html`. Retrieved on December 12, 2017.

[5] Gajawada, A., *Heartbleed bug: How it works and how to avoid similar bugs in the future*, September 6, 2016. `https://www.synopsys.com/blogs/software-security/heartbleed-bug/`. Retrieved on December 12, 2017.

[6] Bart Veldhuizen, *Automated state machine learning of IPsec implementations*, Bachelor thesis, Radboud University Nijmegen, 2017.

[7] Lesly-Ann Daniel, *Inferring OpenVPN State Machines Using Protocol State Fuzzing*, Internship report, University of Rennes 1 and ENS Rennes, 2017.

[8] Paul Fiterau-Brostean, Frits Vaandrager, Erik Poll, Joeri de Ruiter, Toon Lenaerts and Patrick Verleg, *Model Learning and Model Checking of SSH Implementations*, SPIN 2017, p. 142-151, ACM, 2017.

[9] Joeri de Ruiter and Erik Poll, *Protocol state fuzzing of TLS implementations*, USENIX Security, USENIX, 2015.

[10] Janssen, R., *Learning a State Diagram of TCP Using Abstraction*, Bachelor thesis, Radboud University Nijmegen, 2017.

[11] Chow, T., *Testing software design modeled by finite-state machines*, IEEE Transactions on Software Engineering 4, 3 (1978), 178–187.

[12] Beurdouche, B., Bhargavan, K., Fournet, C., Kholweiss, M., Pironti, A., Strub, P., Zinzindohoue, J. K. *A messy state of the union: Taming the composite state machines of TLS*. In Security and Privacy (SP), 2015 IEEE Symposium on (2015), IEEE, pp. 535–552.

[13] De Ruiter, J. *A Tale of the OpenSSL State Machine: a Large-scale Black-box Analysis*, Secure IT Systems: 21st Nordic Conference (NordSec 2016), Lecture Notes in Computer Science, Springer, 2016, Volume 10014, p. 169-184

[14] AlFardan, N., Paterson, K. G. *Lucky thirteen: Breaking the TLS and DTLS record protocols*. 2013.

[15] Kumar, M., *'Freak' - New SSL/TLS Vulnerability Explained*. March 3, 2015. `https://thehackernews.com/2015/03/freak-openssl-vulnerability.html`. Retrieved on December 14, 2017.

[16] Bevilacqua, F., *Finite-State Machines: Theory and Implementation*, October 24, 2013. `https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867`. Retrieved on December 14, 2017.

[17] Angluin, D., *Learning regular sets from queries and counterexamples*. Information and Computation 75, 2 (1987), 87–106.

[18] Bernstein, D. J., *Curve25519: new Diffie-Hellman speed records*, Proceedings of PKC 2006.

[19] Rouse, M., *fuzz testing(fuzzing)*, March, 2010. `http://searchsecurity.techtarget.com/definition/fuzz-testing`. Retrieved on January 11, 2018.

Appendix A

State Machine Diagrams

In this Appendix, all the derived state machines are included. We have two state machines for the OpenSSL 1.1.1 server and two for the WolfSSL 3.13.0 server. In the first and third we only send client input symbols, the second and fourth also contain a selection of server input symbols. Since most of these diagrams might not be readable on paper, we also made them publicly available on `https://gitlab.science.ru.nl/vthoor/bachelorscriptie-2018/tree/master/State%20Machine%20Diagrams`.

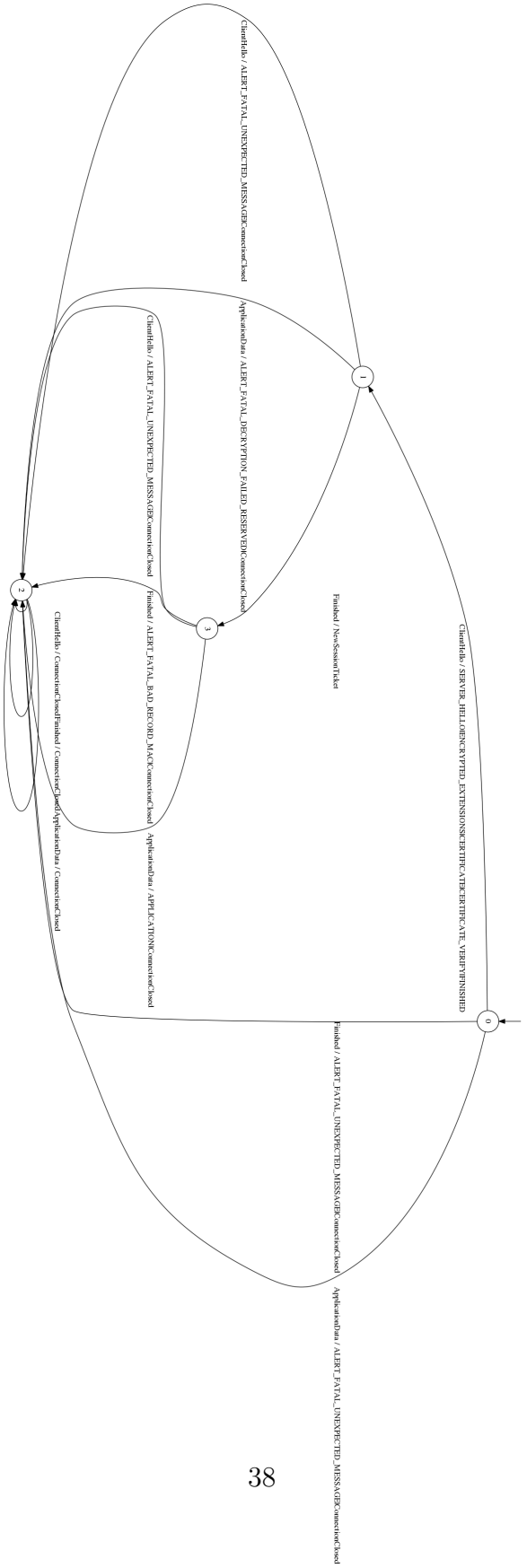

Figure A.1: OpenSSL 1.1.1 server side state machine

Figure A.2: OpenSSL 1.1.1 server side state machine with server input symbols

Figure A.3: WolfSSL 3.13.0 server side state machine

Figure A.4: WolfSSL 3.13.0 server side state machine with server input symbols