BACHELOR THESIS
COMPUTING SCIENCE

RADBOUD UNIVERSITY

# Security analysis of DTLS 1.2 implementations

*Author:*
Niels van Drueten
s4496604

*First supervisor/assessor:*
dr.ir. Joeri de Ruiter
`joeri@cs.ru.nl`

*Second assessor:*
dr. Greg Alpár
`g.alpar@cs.ru.nl`

January 24, 2019

# Abstract

Software libraries such as OpenSSL, mbedTLS and WolfSSL provide a secure transport layer for application data in the network. These libraries secure TCP traffic with the TLS protocol and secure UDP traffic with the DTLS protocol. It is important that the security protocol has been implemented in the right way. A wrong implementation could lead to security vulnerabilities in all applications that use these libraries. This thesis analyses server-side DTLS implementations whether these libraries have implemented the DTLS protocol correctly by using state machine inference. Using this technique, we can create a state machine by only knowing the input and output alphabet of the DTLS implementation. We were able to create and analyse the state machine for OpenSSL and mbedTLS. This thesis found no security vulnerabilities in OpenSSL, but found unexpected behaviour in mbedTLS. We were not able to create a state machine for WolfSSL.

# Contents

1 Introduction 2

2 Preliminaries 4
- 2.1 Basics of TLS 1.2 . . . . . 4
- 2.2 DTLS 1.2 . . . . . 12
  - 2.2.1 Timeout and Retransmission . . . . . 12
  - 2.2.2 Fragmentation . . . . . 13
  - 2.2.3 Handshake header modification . . . . . 14
  - 2.2.4 Stateless cookie exchange addition . . . . . 15
- 2.3 State machine inference . . . . . 16

3 Research 21
- 3.1 Tools . . . . . 21
  - 3.1.1 TLS-Attacker . . . . . 21
  - 3.1.2 TLS-Attacker-Connector . . . . . 22
  - 3.1.3 Statelearner . . . . . 24
- 3.2 Research set-up . . . . . 25
- 3.3 OpenSSL . . . . . 26
- 3.4 mbedTLS . . . . . 29
- 3.5 WolfSSL . . . . . 33
- 3.6 Implementation fingerprinting . . . . . 33

4 Related Work 35

5 Conclusions 36

A Appendix 41

# Chapter 1

# Introduction

Applications that would like to communicate over the internet will likely use transport layer protocols. The two main transport protocols are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) [1] [2]. TCP excels in reliability whereas UDP excels in speed. TCP is the most common transport layer protocol. However, UDP is being used by more and more applications [3]. Applications that require a fast connection between client and server prefer UDP over TCP. TCP provides reliable data transfer. This means that when packets arrive in the wrong order, it reorders the packets. When packets do not arrive at all, it retransmits. This results in more overhead. UDP does not provide reliable data transfer. This makes UDP faster than TCP, but less reliable.

Both transport protocols are not secure by themselves. An adversary in the network can intercept packets and read or modify data that is being sent. The most common way to secure a TCP connection is by using the Transport Layer Security (TLS) protocol. As the RFC of TLS states, TLS provides privacy and data integrity between two communicating applications [4]. To secure UDP traffic, Datagram Transport Layer Security (DTLS) has been published [5] [6] [7].

DTLS is a security protocol based on the TLS protocol and provides the same communication privacy as TLS. It is important that the implementation of the DTLS protocol is done in the right way. A wrong implementation could lead to a vulnerability in the software. This vulnerability could be exploited by an adversary and secure communication will become insecure. The adversary will be able to read the application data. Software libraries that provide secure communication such as OpenSSL, mbedTLS and WolfSSL, also have DTLS support. Applications could use these libraries for their application communications. A vulnerability in one of these libraries, or in other implementations of DTLS in general, result in a vulnerability in all applications that use these libraries. Research in these protocol implementations is vital to detect and remove vulnerabilities.

In this thesis, we analyse several server-side DTLS implementations by using state machine inference. This technique uses a black-box approach to create a state machine by only knowing the input and output alphabet of the implementation. Analysing the end result of the state machine, we can conclude whether the server-side DTLS implementation has vulnerabilities. If we do not find a vulnerability in the implementation, it does not mean that the implementation has no vulnerabilities. It could still be vulnerable to other attacks, for example buffer overflow or man-in-the-middle attacks.

This thesis explains the basics of TLS 1.2 in section 2.1, how DTLS works in section 2.2, what state machine inference is in section 2.3. The tools we use in section 3.1, how we have analysed server-side DTLS implementations in section 3.2 as well as the results in section 3.3, 3.4 and 3.5. Related work will be discussed in section 4. This thesis will end with a conclusion in section 5.

# Chapter 2

# Preliminaries

In this section we explain the information needed to understand this thesis. The basics of TLS 1.2 will be explained in section 2.1. The DTLS 1.2 protocol will be explained in section 2.2. In section 2.3, the technique of state machine inference will be explained.

## 2.1 Basics of TLS 1.2

DTLS 1.2 is based on TLS 1.2 with some differences. The internals of DTLS 1.2 are the same as TLS 1.2. However, to cope with the unreliability of UDP, DTLS has to implement a solution for packet loss and packet reordering. Secondly, Denial-of-Service (DOS) possibilities arise while UDP is connectionless. In August, TLS 1.3 has been released. However, DTLS 1.3 is still work in progress [7]. Since DTLS 1.3 is still under development, many libraries do not have DTLS 1.3 support. Therefore we are analysing DTLS 1.2 implementations. To understand DTLS 1.2, we first need to learn how TLS 1.2 works. RFC 5246 specifies TLS 1.2 [4].

T. Dierks and C. Allen specified TLS 1.0 in January 1999 [8]. In April 2006, T. Dierks and E. Rescorla improved TLS 1.0 to TLS 1.1. TLS 1.1 has some small security and clarification improvements [9]. TLS 1.2 has been specified in August 2008 [4]. The biggest improvement in this version are flexibility in cipher suite negotiation. In August 2018, TLS 1.3 has been specified by E. Rescorla [10].

A TLS 1.2 connection starts with a TLS handshaking protocol between client and server. The handshaking protocol has three subprotocols: Handshake Protocol, Change Cipher Spec Protocol and the Alert Protocol.

- Handshake Protocol:
  This protocol handles the cryptographic configuration that will be used during and after the handshake. The handshake protocol has the following goals as stated in section 7.3 of RFC 5246: “A client and server agree on a protocol version, select cipher suites, optionally

authenticate each other, and reliable negotiation of a shared key takes place" [4].

- Change Cipher Spec Protocol:
  The Change Cipher Spec protocol handles a change in the cipher suite. TLS uses this protocol, when a client or server changes the cryptographic algorithm. All messages after the ChangeCipherSpec message use the new cryptographic algorithm. This means that after the ChangeCipherSpec message, both sides start encrypting and decrypting by using the computed keys.

- Alert Protocol:
  The Alert protocol consists of warning and fatal error messages. When a peer receives a fatal error message, the connection must be terminated immediately. For example, a client could receive an unexpected fatal error message in the handshake protocol. This means that the server has received an unexpected message from the client. The message flow in the handshake protocol is strict. Whenever an unexpected message has been received by either the client or server, the handshake must fail. Receiving an unexpected fatal error message means that the implementation of the protocol is not correct. A correctly implemented client and server should never receive an unexpected message error when communicating.

  A warning error message could be a user cancellation or receiving an unknown/expired/revoked certificate. These warnings are not fatal. In theory, a connection could continue when an expired certificate is received. In TLS 1.3 this is not possible anymore. Every error message will result in a failed handshake.

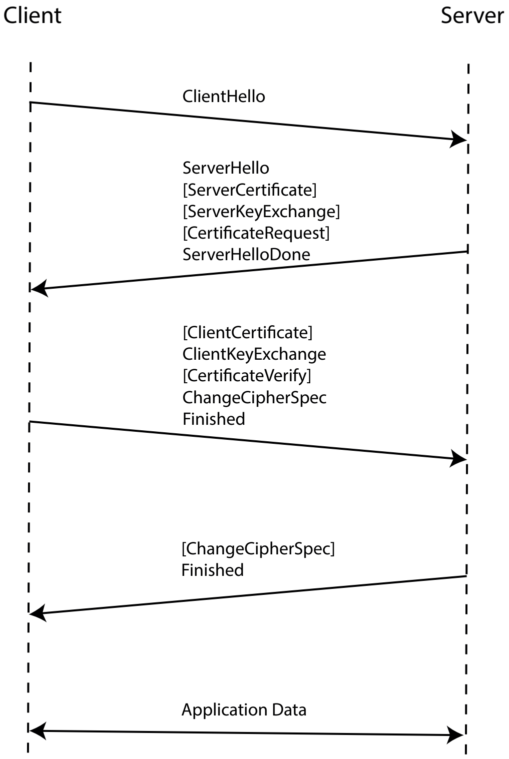

Figure 2.1: TLS 1.2 handshake message flow

Figure 2.1 shows the strict handshake message flow between client and server. We explain each message below. Handshake messages between closed brackets are situational messages. These messages are necessary in some situations, depending on the cipher suite that will be used.

1. **ClientHello**: (required)
   The client starts the handshake by sending the ClientHello to the server. The main goal for this message is to communicate cipher suites and compression algorithms it would like to speak with the server. The client creates a list of supported cipher suites and compression algorithms. We will use the cipher suite `TLS_RSA_WITH_AES_128_CBC_SHA256`. This cipher suite contains:

   - `RSA`: Used for key exchange.
   - `AES_128_CBC`: Used for symmetric encryption.
   - `SHA256` Used for hashing.

   A client could start a new session or resume a session. The ClientHello messages includes a session ID. If the session ID is empty, the client would like to start a new session. If the session ID is not empty, the client would like to resume a connection.

The ClientHello also includes a random value. This random value is randomly generated by the client which will be used to compute the `master_secret` later on. The last thing present in the ClientHello are extensions. A client could request additional functionality with extensions. If the server is not able to handle this additional functionality, the client may abort the connection.

An example of a ClientHello extension is the signature algorithm extension. This extension requests the server to add signatures in their ServerCertificate. The certificate list in the ServerCertificate message must be signed using this signature algorithm.

After sending a ClientHello message, the client waits until it receives a ServerHello message.

2. **ServerHello**: (required)
   After the server received the ClientHello message, the server answers with a ServerHello. In this message the server takes one cipher suite and one compression algorithm out of the list in the ClientHello message. Now, the client and server have agreed on which cipher suite to use during and after the handshake. If the server cannot find a supported algorithm, the handshake fails with an handshake failure error. This error is fatal, so the handshake must be terminated.

   If client and server have already communicated with each other, they could resume the session using the session ID in the Hello messages. If the session ID is not empty, the server looks in its cache for a match. If a match has been found, the ServerHello message uses the same session ID to indicate a previous session will be continued now. The cipher suite and compression algorithm will be the same as the last session. In this case the handshake will continue with the Finished message, which will be explained later.

   Just as the ClientHello, the ServerHello message also contains a random value. This value is randomly generated by the server and does not have any relation to the clientHello random value. Both random values will be used to compute the `master_secret`.

3. **ServerCertificate**:
   After the ServerHello message, the server sends another message to the client: the ServerCertificate message. This message is not required in every situation, but in most cases it is. If the key exchange uses certificates for authentication, this message is required. Cipher suites with RSA key exchange use RSA certificates, cipher suites with Diffie-Hellman (DH) key exchange use certificates with DH parameters and cipher suites with ephemeral Diffie-Hellman (DHE) uses certificates with DHE parameters. In this message the server sends its certificate

to the client to authenticate itself to the client. Some cipher suites use anonymous DH key exchange. When using this cipher suite, neither the client nor the server authenticates itself. This cipher suite makes TLS vulnerable to man-in-the-middle attacks.

This message contains a list of certificates, starting with the server's certificate and ending with root certificate authority.

**Certificate verification**:
The client can verify whether it is communicating with the real server by using this certificate list. The client should start by verifying the root authority certificate. Then, the client verifies the next certificate in the list with the parent certificate. The client iterates this method until the client has reached the servers's certificate which is the last on the list.

4. **ServerKeyExchange**:
   If the ServerCertificate does not contain enough data to exchange a premaster secret, the server sends a ServerKeyExchange message. This message is only required when using a ciphersuite with DHE_DSS, DHE_RSA or DH_anon key exchange. It is forbidden to send a ServerKeyExchange message when using a ciphersuite with RSA, DH_DSS or DH_RSA. When using a DHE key exchange, this message contains the prime modulus ($p$), generator ($g$) and public key ($g^X \mod p$), where X is the private key of the server.
5. **CertificateRequest**:
   A server could request a certificate from the client with this message. The client must authenticate itself in the next message with the ClientCertificate message. Only an authenticated server is allowed to request a client's certificate. If an anonymous server request sends this message, the handshake should fail with a fatal error message.
6. **ServerHelloDone**: (required)
   This message indicates the end of the server messages at this point. The server will wait for a response of the client. This message does not contain any data.
7. **ClientCerticate**:
   If the server has requested a client certificate with the CertificateRequest message, the client should respond with a ClientCertificate message. If the client does not have a client certificate, this message has length zero. The server will decide whether it will continue the handshake.
8. **ClientKeyExchange**: (required)
   In this message the client sends a premaster secret to the server. The

data in this message depends on what cipher suite is selected earlier in the handshake. After this message the client and server will be able to construct a master key. When using a ciphersuite with RSA key exchange, this message contains the 48-byte premaster secret computed by the client. The client is able to compute the premaster secret, with the data the server provided in the ServerCertificate and ServerKeyExchange. The premaster secret will not be send in plain text, this would be insecure. It will be encrypted with the public key of the server. Only the server will be able to decrypt the premaster secret. Now, the client and server have a shared premaster secret. This secret will be used later on when client and server compute the `master_secret`. When using a ciphersuite with DH key exchange, the client's public value will be send to the server. Both parties have the public value of itself and the other party and can compute the DH `master_secret`. This `master_secret` will be used after the ChangeCipherSpec and the Finished message is the first encrypted message using this `master_secret`.

**Computing the master_secret**:
The `master_secret` is computed with the pre-master secret, client random value and server random value. Both client and server will compute the `master_secret` with a Pseudo Random Function (PRF):

```
master_secret = PRF(pre_master_secret, ''master secret'',
ClientHello.random + ServerHello.random)
```

The PRF can produce arbitrary lengths output. It has 3 parameters, namely the secret, label and seed and is defined as:

```
PRF(secret, label, seed) = P_<hash>(secret, label + seed)
```

The hash function, used in the PRF and Keyed-Hashing for message Authentication (HMAC), is part of the cipher suite, which has been chosen in the Hello messages. In the cipher suite `TLS_RSA_WITH_AES_128_CBC_SHA256`, SHA256 will be the hash function used in the PRF and HMAC.

`P_<hash>(secret, label + seed)` is defined as:

```
P_<hash>(secret, label + seed) =
HMAC_hash(secret, A(1) + seed) +
HMAC_hash(secret, A(2) + seed) +
HMAC_hash(secret, A(3) + seed) + ...
```

Where A is defined as:

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

The `HMAC_hash()` results in a byte length of 32. To compute the `master_secret` with 48 bytes, we need to run `HMAC_hash()` twice. Where the last 16 bytes are not used.

9. **CertificateVerify**:
   This message is sent by the client when the certificate has signing capability. This message provides explicit verification of the client's certificate.

10. **ChangeCipherSpec**: (required)
    The ChangeCipherSpec message is used during the handshake, but it is not part of the handshake protocol. This message signals that all messages after the ChangeCipherSpec message will use the new cryptographic algorithm and `master_secret`. Both sides start encrypting and decrypting using the computed keys.

    **Key derivation**:
    The `master_secret` will be used to compute a key_block, from where the following keys are derived:

    - client_write_MAC_key
    - server_write_MAC_key
    - client_write_key
    - server_write_key

    The key_block is computed with:

    ```
    key_block = PRF(master_secret, ‘‘key expansion’’,
    server.random + client.random)
    ```

    The first block of bits with length client_write_MAC_key is the client_write_MAC_key, the next block of bits with length server_write_MAC_key is the server_write_MAC_key et cetera. The server and client are using distinct symmetric keys to encrypt their data. As a server, it is not able to echo a client's message back to the client. The client will not be able to decrypt it, because it uses a different key to decrypt. To echo a message, the server has to decrypt the message with the client_write_key and encrypt the message with the server_write_key.

11. **Finished**: (required)
    The Finished message is the first encrypted message between client and server using the shared `master_secret`. This message always

follows after a ChangeCipherSpec message. The server should be able to decrypt this message and can confirm the integrity of the handshake and that client and server have the same shared key. After this message the client and server can exchange application data.

The ChangeCipherSpec and Finished message is sent by both parties at the end of the handshake. It contains a `verify_data` 96-byte value.

- If this message is sent by a server, the `verify_data` will be computed by the following:
  `verify_data = PRF (master_secret, ‘‘server finished’’, hash(handshake_messages))`.
- If this message is sent by a client, the `verify_data` will be computed by the following:
  `verify_data = PRF (master_secret, ‘‘client finished’’, hash(handshake_messages))`.

Again, the PRF is used here. One of the PRF parameters is a hash of handshake_messages. This hash is chosen by the cipher suite that is being used. The handshake_messages is a list of all handshake messages excluding this Finished message.

After the TLS handshake, the client and server have a protected connection with symmetric keys derived from the `master_secret`. One symmetric key is for client message encrypting/decrypting and one is for server message encrypting/decrypting.

### TLS 1.3

TLS 1.3 has removed insecure cryptographic algorithms, such as DES and MD5. Next to this, all handshake messages after the ServerHello are encrypted. The last major difference between TLS 1.3 and TLS 1.2 is that the handshake is 1 round trip shorter, this results in a quicker handshake. TLS 1.3 has a so called 0-RTT (zero round-trip time), which is even faster. This new feature allows clients to send more data in the first message and reduces the handshake round-trips. This is only possible when client and server already share a Pre-Shared Key. Client and server could obtain a Pre-Shared Key externally or via a previous handshake. The shorter 0-RTT handshake is less secure than the normal handshake. Appendix E.5 of the RFC states that the 0-RTT handshake is vulnerable to replay attacks. An adversary could replay a previous 0-RTT handshake and duplicate the action that has been taken. Buying an item in a webshop or transferring money could be duplicated when TLS 1.3 uses the 0-RTT handshake. To protect a customer from a replay attack, a server has to prevent these attacks by implementing one of the methods described in section 8 of RFC8446. One of the suggested methods would be that the server does not allow a session ID to be used more than once. A session ID that already has been used, must be denied by the server. If the 0-RRT hanshake fails, TLS 1.3 will fall back to the standard 1-RTT handshake. This feature is not available in TLS 1.2 and thus not available in DTLS 1.2.

## 2.2 DTLS 1.2

As explained in the introduction, it is not possible to run TLS over UDP as TLS requires a reliable transport layer [4]. UDP does not provide a reliable transport layer. TLS will break when it runs over UDP when a packet is lost or reordered. Therefore DTLS has been made to secure UDP traffic. This section is mostly based on RFC 6347, which explains the DTLS 1.2 protocol [6].

The Datagram Transport Layer protocol (DTLS) is a security protocol on top of the User Datagram Protocol (UDP). DTLS provides secure network traffic as stated in the RFC of DTLS 1.2. DTLS has been designed in such a way that it is similar to TLS and can run over UDP to create a fast and secure connection between client and server. DTLS uses the same handshake messages as TLS. Minimizing security invention and reusing code and infrastructure are the main reasons to build DTLS in the same way as TLS [6]. As DTLS provides communication privacy the same way as TLS, DTLS does not reinvent the wheel. The code base of the TLS implementation can be reused when implementing DTLS. Closing a DTLS connection the implementation should send a close_notify alert. This way, the peer knows the connection is ending.

DTLS 1.0 is based on TLS 1.1, DTLS 1.2 is based on TLS 1.2 [5] [6]. On November 5, 2018 a draft has been published for DTLS 1.3 that is designed based on TLS 1.3 [7]. TLS 1.3 has been approved in August 2018 [10]. To run parallel with TLS versions, DTLS 1.1 has been skipped.

This section explains how DTLS 1.2 solves unreliability and what has changed compared to TLS 1.2.

Unreliability, message size and Denial-of-Service attack possibilities are problems DTLS has to solve. The unreliability problem could be divided in to two problems, packet loss and packet ordering. In the next section we will separately discuss the four problems and how DTLS solves these.

### 2.2.1 Timeout and Retransmission

The first problem is packet loss. The TLS handshake assumes no packets are lost. With UDP, there is a possibility that packets get lost. To solve the packet loss problem, DTLS uses a simple retransmission mechanism. Both sides, client and server, use a timer to keep track how long it takes to receive a response. If, for example, the timer expires on the client side, it will retransmit its last packet sent. A time could expire because its own message got lost or the response message got lost. In both cases the client will retransmit.

DTLS uses a state machine for retransmissions. There are four states in the state machine: PREPARING, SENDING, WAITING and FINISHED.

- PREPARING:
  During the PREPARING state, the client or server creates messages.
- SENDING:
  During the SENDING state, the client or server sends the messages to the peer.
- WAITING:
  During the WAITING state, the client or server waits for a response from the peer. In this state, client or server starts a timer. When the timer expires it will enter the SENDING state again and retransmit the last messages. Receiving a message from its peer will stop the timer.
- FINISHED:
  When the handshake is done, the implementation enters the FINISHED state and application data will be sent. To keep the fast UDP properties no retransmission will take place in this state.

The timer length depends on the amount of retransmission that already have been done. The default start timer length is 1 second. After every retransmission the timer length doubles, with a maximum of 60 seconds. Doubling the timer length prevents more congestion on the connection between client and server. If the connection is already congested, a constant timer length would congest the connection even more. The timer will reset to 1 second once a transmission has been sent without loss or after an idle period of 10 times the current timer value. If the current timer value is 60 seconds, the timer will reset to 1 second after a successful transmission or after 600 seconds.

### 2.2.2 Fragmentation

The second problem to solve is the message size. TCP packets could have a maximum of $2^{24} - 1$ bytes. When a TCP message is too large, the IPv4 protocol fragments it. TCP provides a byte stream for TLS, so TLS does not have to worry about fragmentation. However, a UDP datagram without IP fragmentation could only be 1500 bytes. The TLS handshake messages that are more than 1500 bytes needs to be fragmented. The fragments will be sent in separate datagrams. To send a TLS handshake message in DTLS, one has to create a TLS handshake message, fragment the handshake message into smaller fragments and send all fragments to its peer. The peer will combine the fragments to a TLS handshake message. The fragments needs to contain a fragment length and a fragment offset in order to combine the fragments in the right order. The fragment length indicates the length of the data in this fragment, the fragment offset indicates the length of the data in all previous fragments combined.

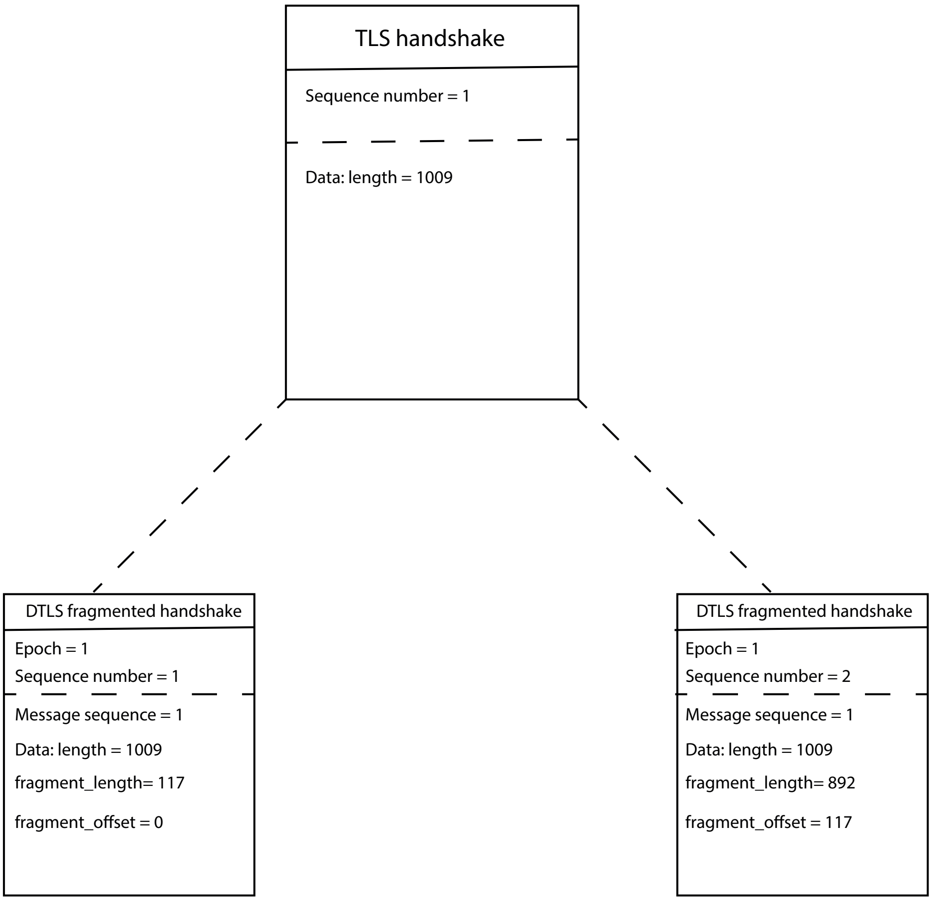

Figure 2.2: An example of DTLS 1.2 fragmentation

As shown in figure 2.2, there are two sequences numbers and an epoch, this will bring us to the next DTLS difference.

### 2.2.3 Handshake header modification

The third problem is packet ordering. With UDP, packets could arrive in the wrong order. Therefore, DTLS should reorder the packets itself. To handle this, DTLS adds an epoch number and a second sequence number to the header of a handshake message. The second sequence number is called message sequence. The newly added message sequence and epoch are located in the header of a DTLS record layer.

The epoch value starts at 0 and will increase after every cipher state change. It will increase to 1 in the Finished message, because the Finished message is encrypted with a different cipher state. This epoch value helps the implementation to decide with which cipher state this packet is encrypted. It is possible that the message of a new cipher state arrives earlier than the cipher state announcement. Without the epoch number, the implementation would decrypt the message with the wrong cipher state. With the epoch number, the implementation knows it belongs to a different cipher state.

The sequence number is used to verify the TLS MAC. It starts at 0 and increases with every message. When the epoch is increased, the sequence number starts at 0 again.

The sequence number and message sequence are increasing linearly, until fragmentation takes place. As seen in figure 2.2, the sequence number of the fragmented messages increases with every fragmentation. However, the message sequence stays the same. The message sequence is used to combine all fragmented message into one message. After the fragmentation, the sequence number and message sequence increase linearly again.

### 2.2.4 Stateless cookie exchange addition

The last problem is the possibility of a Denial-of-Service attack. An adversary could start multiple handshake requests and causing the server to allocate memory and performing cryptographic operations. This will slow or even disrupt other connections. Therefore causing a denial-of-service attack (DoS). A second DoS attack is an amplification attack. In this attack, the attacker sends a small ClientHello message to the server and receives a large CertificateMessage back. Spoofing the source IP address to a victim's IP address, the unknowing DTLS server will send a large Certificate message to the victim's IP address. The victim gets flooded by the innocent server.

DTLS 1.2 adds a stateless cookie exchange to the TLS handshake to prevent denial-of-service attacks. This stateless cookie exchange happens in the Hello messages. Just as TLS 1.2, the client starts a handshake with a ClientHello message. In DTLS 1.2, the server will not respond with a ServerHello immediately, but responds with a new message which we have not seen in TLS 1.2. This new message is called **HelloVerifyRequest**. The HelloVerifyRequest message consist of a cookie generated by the server. The cookie is computed with the following computation:
`Cookie = HMAC(Secret, Client-IP, Client-Parameters)` The secret is a server-side secret which should be changed frequently. Keeping the secret for a long time, a new attack is possible. An adversary can collect several cookies from different IP addresses and reuse them later. Changing the secret will invalidate all previous cookies.

After the client has received the HelloVerifyRequest, it will send the same ClientHello message again, but with the cookie added. The server will verify if the second ClientHello contains the right cookie and will continue with the ServerHello.

This DoS prevention costs an extra flight in the handshake and thus increases the handshake time.

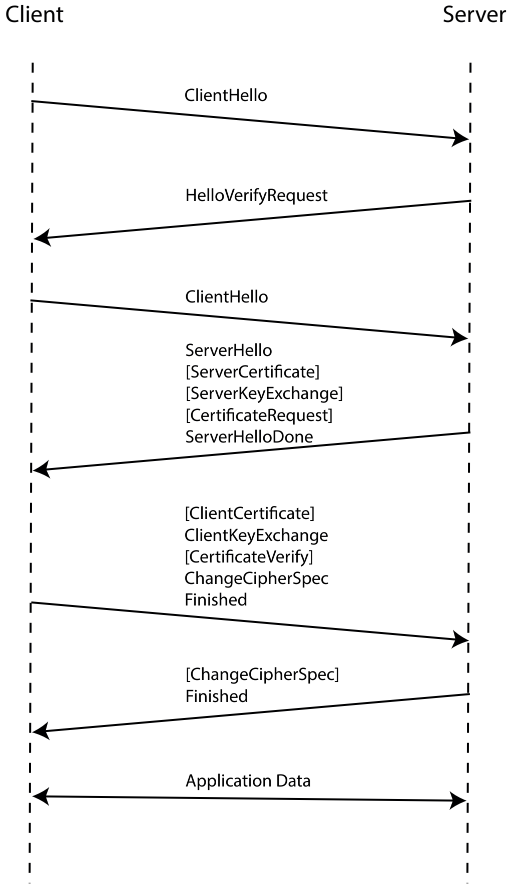

Figure 2.3: DTLS 1.2 handshake message flow

Figure 2.3 shows a DTLS 1.2 handshake message flow. Comparing this to the TLS 1.2 handshake message flow in figure 2.1, we can see that DTLS 1.2 uses an extra flight in the Hello messages.

## 2.3 State machine inference

In this thesis, we are using Mealy machines to learn the different states of a server-side DTLS implementation. A Mealy machine is a deterministic finite-state machine, where the state and input determine the output [11]. A Mealy machine is a 6-tuple $(S, S_0, \Sigma, \Lambda, T, G)$ where:

- $S$: Finite set of states.

- $S_0$: Start state, where $S_0 \in S$.
- $\Sigma$: Finite set of input alphabet.
- $\Lambda$: Finite set of output alphabet.
- $T$: State transition function $T : S \times \Sigma \to S$.
- $G$: Output function $G : S \times \Sigma \to \Lambda$.

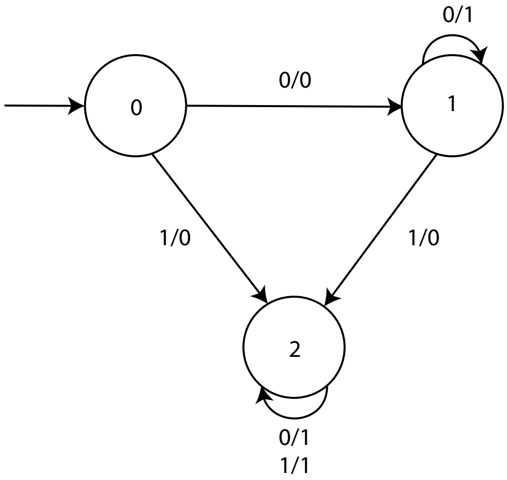

Figure 2.4: Example of a Mealy machine

Figure 2.4 is an example of a Mealy machine. This Mealy machine has the following 6-tuple:

- $S = \{0, 1, 2\}$
- $S_0 = 0$, where $0 \in S$.
- $\Sigma = \{0, 1\}$
- $\Lambda = \{0, 1\}$
- $T$: State transition function $T : S \times \Sigma \to S$:
  Represented in a state transition table:

| | State 0 | State 1 | State 2 |
|---|---|---|---|
| Input 0 | State 1 | State 1 | State 2 |
| Input 1 | State 2 | State 2 | State 2 |

- $G$: Output function $G : S \times \Sigma \rightarrow \Lambda$:
  Represented in an output table:

| | State 0 | State 1 | State 2 |
|---|---|---|---|
| Input 0 | Output 0 | Output 1 | Output 1 |
| Input 1 | Output 0 | Output 0 | Output 1 |

The start state, $S_0$ is indicated with an arrow ($\rightarrow$). Starting in $S_0$ with an input string 01, results in a state transition from $S_0 \rightarrow S_1$ and an output string of 01 following the state transition table and the output table. The Mealy machine is deterministic, because for each input and state, it always ends in the same state, with the same output. It is also a finite-state machine, because the state machine has finite states and finite state transitions.

State machine inference creates a Mealy machine from a protocol implementation. Using a black-box approach, it is possible to create a state machine by only knowing the input and output alphabet. This will be the set of protocol messages. States, state transitions and output values are learned during the process.

Two algorithms are used when performing state machine inference. A learning algorithm and an equivalence algorithm:

- **Learning algorithm**:
  This algorithm learns the protocol it is talking to. By sending protocol messages, state machine inference can learn the states, state transitions and output of the protocol implementation.

  At the end of the learning algorithm, a hypothesis is made and this hypothesis will be sent to the equivalence algorithm.

- **Equivalence algorithm**:
  With the hypothesis of the learning algorithm, the equivalence algorithm tests whether the hypothesis represents the actual protocol implementation. If it does not represent the actual protocol implementation, a counter-example will be made and sent to the learning algorithm. The learning algorithm starts again. This will go on, until a hypothesis is accepted that represents the protocol implementation.

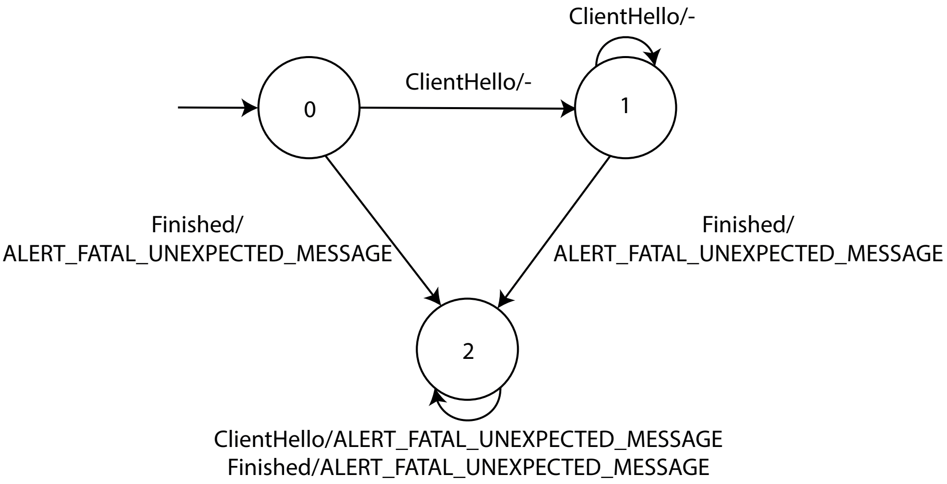
ClientHello/-
ClientHello/-
0
1
2
Finished/
ALERT_FATAL_UNEXPECTED_MESSAGE
Finished/
ALERT_FATAL_UNEXPECTED_MESSAGE
ClientHello/ALERT_FATAL_UNEXPECTED_MESSAGE
Finished/ALERT_FATAL_UNEXPECTED_MESSAGE

Figure 2.5: A practical example of a Mealy machine

Figure 2.5 is an example, with a closer input/output alphabet to our actual problem. This Mealy machine has an input of TLS protocol messages, and an output of TLS protocol messages, including errors:

- $S = \{0, 1, 2\}$
- $S_0 = 0$, where $0 \in S$.
- $\Sigma = \{$ ClientHello, Finished $\}$
- $\Lambda = \{$ -, ALERT_FATAL_UNEXPECTED_MESSAGE $\}$
- $T$: Transition function $T : S \times \Sigma \rightarrow S$:
  Represented in a state transition table:

| | State 0 | State 1 | State 2 |
|---|---|---|---|
| ClientHello | State 1 | State 1 | State 2 |
| Finished | State 2 | State 2 | State 2 |

- $G$: Output function $G : S \times \Sigma \rightarrow \Lambda$:
  Represented in an output table:

| | State 0 | State 1 | State 2 |
|---|---|---|---|
| ClientHello | - | - | ALERT_FATAL_ UNEXPECTED_MESSAGE |
| Finished | ALERT_FATAL_ UNEXPECTED_MESSAGE | - | ALERT_FATAL_ UNEXPECTED_MESSAGE |

With an input algorithm of $\Sigma = \{$ ClientHello, Finished $\}$ , the state machine inference can start learning this example protocol.

Starting in the starting state $S_0$, the learning algorithm starts sending the input alphabet and will learn this simple implementation.

- **Sending ClientHello**:
  The algorithm learns a new output `-`,
  creates a new state $S_1$,
  with a new state transition $S_0 \rightarrow S_1$.
- **Sending Finished**:
  The algorithm learns a new output `ALERT_FATAL_UNEXPECTED_MESSAGE`,
  creates a new state $S_2$,
  with a new state transition $S_0 \rightarrow S_2$.

Next round:

- **Sending ClientHello/ClientHello**:
  The algorithm learns a new state transition $S_1 \rightarrow S_1$.
- **Sending ClientHello/Finished**:
  The algorithm learns a new state transition $S_1 \rightarrow S_2$.
- **Sending Finished/ClientHello**:
  The algorithm learns a new state transition $S_2 \rightarrow S_2$.
- **Sending Finished/Finished**:
  The algorithm learns a new state transition $S_2 \rightarrow S_2$.

This state machine will be checked by the equivalence algorithm whether it represents the actual protocol. If this check passes, the algorithm is finished. If it did not pass the test, the learning algorithm tries to create a new state machine.

# Chapter 3

# Research

In this section, we explain the tools we have used. After that, the research set-up will be explained. The analysis and the result from each server-side implementation of the DTLS protocol are discussed separately.

## 3.1 Tools

The tools we have used are **TLS-Attacker**, **TLS-Attacker-Connector** and **Statelearner**.

### 3.1.1 TLS-Attacker

TLS-Attacker is a framework to analyze TLS implementations developed by the Ruhr University Bochum and the Hackmanit GmbH written in Java 8 [12]. It can send TLS messages to the implementation in a random order. This way it tries to break the implementation. TLS-Attacker supports TLS 1.0, TLS 1.1, TLS 1.2 and TLS 1.3. A feature to analyze DTLS implementations is being built. This feature is still unstable and has yet to be released in the public repository. The feature is almost done and together with the developers, Joeri de Ruiter, Paul Fiterau Brostean and I have debugged this feature to a working DTLS feature. TLS-Attacker is able to complete a handshake and send application data to a DTLS server, but it is still unstable. Receiving a retransmission will fail the handshake. Future work has to be done in order to handle retransmission. TLS-Attacker is available at `https://github.com/RUB-NDS/TLS-Attacker`. Compiling the project is done by Maven, with the command:

```
$ mvn clean install
```

Installing Maven on a Ubuntu machine, can be done by:

```
$ sudo apt-get install maven
```

To test the TLS-Attacker client, a TLS server is required. We advise to run a TLS server locally as TLS-Attacker could shutdown the server. While running a TLS server locally on TCP port 4432, one could start a TLS-Attacker client with the following command:

```
$ java -jar TLS-Client.jar -connect localhost:4432
```

This will start performing a TLS handshake with a TLS server on localhost:4432. To test a handshake with a DTLS server on UDP port 4433, use the following command:

```
$ java -jar TLS-Client.jar -connect localhost:4433
-version DTLS12
```

This will start a TLS handshake with a DTLS server on localhost:4433.

It is also possible to run TLS-Attacker as a library, which you can import in the project and create your own TLS message flow. Therefore you will need a Config class. This class stores all configurations of TLS-Attacker, such as server IP address, server port, protocol version and a list of cipher suites. The WorkflowTrace class is used to create a TLS message flow, this could be different from a normal handshake. In the example below the program sends a ClientHelloMessage and expects to receive a ServerHelloMessage back. The State class stores the configuration of the program and the message flow, which will be executed with the DefaultWorkflowExecutor.

```
Config config = Config.createConfig();
WorkflowTrace trace = new WorkflowTrace();
trace.addTlsAction(new SendAction(new ClientHelloMessage()));
trace.addTlsAction(new ReceiveAction(new ServerHelloMessage()));
State state = new State(config, trace);
DefaultWorkflowExecutor executor = new
DefaultWorkflowExecutor(state);
executor.executeWorkflow();
```

In the next section we will use this to create the TLS-Attacker-Connector.

### 3.1.2 TLS-Attacker-Connector

TLS-Attacker-Connector is a program that translates message strings to WorkflowTrace for TLS implementations classes made by Joeri de Ruiter [13]. I have extended this program with a DTLS feature. The message strings are used by Statelearner to learn a TLS/DTLS implementation. TLS-Attacker-Connector translates these message strings into XML strings that are used by TLS-Attacker to execute a handshake message. TLS-

Attacker-Connector also receives messages from TLS-Attacker and translates these messages into string messages and sends it to Statelearner.

TLS-Attacker-Connector is available at `https://gitlab.science.ru.nl/joeri/TLS-Attacker-Connector`. The code for DTLS 1.2 is available on the DTLS12 branch. To start TLS-Attacker-Connector, use the following command:

```
$ java -jar target/TLSAttackerConnector2.0.jar -pV DTLS12.
```

This creates a Config class, with the following configurations:

- Ciphersuite = TLS_RSA_WITH_AES_128_CBC_SHA256.
- Compressionmethod = NULL
- ProtocolVersion = DTLS12

By default, TLS-Attacker-Connector targets a server on port 4433. If you would like to change the port, you can change it with the -tP <port> flag.

To reset a TLS connection, it was enough to reinitialize the transport handler. This was not the case for DTLS. To reset a DTLS connection, we have to stop the server and start the server again. Without restarting the server, we were not able to perform a second handshake after a successful handshake with TLS-Attacker and the DTLS server. We have added an option to start an external process. This external process starts a DTLS server. To reset the UDP connection, we only had to stop the external process and start it again. The -eS <process> flag, creates an external process <process>. The supported external processes are OpenSSL, mbedTLS and WolfSSL.

With the --test flag, TLS-Attacker-Connector starts a default handshake with the server. When using TLS, it will send the ClientHello, RSAClientKeyExchange, ChangeCipherSpec, Finished, ApplicationData and AlertWarningCloseNotify. When using DTLS, it will send the ClientHello, ClientHello, RSAClientKeyExchange, ChangeCipherSpec, Finished, ApplicationData and AlertWarningCloseNotify.

| **Message String** | **Action** |
|---|---|
| AlertWarningCloseNotify | Sends AlertWarningCloseNotify message |
| ApplicationData | Sends ApplicationData message |
| ChangeCipherSpec | Sends ChangeCipherSpec message |
| ClientHello | Sends ClientHello message |
| Finished | Sends Finished message |
| RESET | Resets connection with server |
| RSAClientKeyExchange | Sends RSAClientKeyExchange message |

Table 3.1: TLS-Attacker-Connector message strings

Using the RESET string, TLS-Attacker-Connector will reset the connection with the server. When using TCP, TLS-Attacker-Connector will restart the connection. When using UDP, TLS-Attacker-Connector will restart the external process.

### 3.1.3 Statelearner

Statelearner is a tool for state machine inference made by Joeri de Ruiter. It can learn state machines from implementations using a block-box approach [14]. It uses LearnLib for the learning and equivalence algorithms [15]. Providing Statelearner with a predefined alphabet, it will learn states of the implementation. The predefined input alphabet should be listed in the configuration file of Statelearner as well as the hostname nand port of TLS-Attacker-Connector. Statelearner does not have information about the implementation it is talking to. Using the output of the implementation it tries to learn the implementation and create a Mealy state machine. We analyse the created state machine of each implementation in section 3.3 and 3.4.

The `examples/socket/socket.properties` is an example configuration file for TLS/DTLS implementations. The hostname and port should be pointing to the TLS-Attacker-Connector, which is listening on localhost:6666. The alphabet is a list of message strings Statelearner should use, seperated with a space.

The configuration file `dtls.properties` we use has the following configurations:

- **type**: socket
- **hostname**: localhost
- **port**: 6666
- **alphabet**: ClientHello RSAClientKeyExchange ChangeCipherSpec Finished ApplicationData AlertWarningCloseNotify

- **output_dir**: output
- **learning_algorithm**: lstar
- **eqtest**: randomwords
- **min_length**: 5
- **max_length**: 10
- **nr_queries**: 100
- **seed**: 1

Statelearner will only use client messages in our analysis.

## 3.2 Research set-up

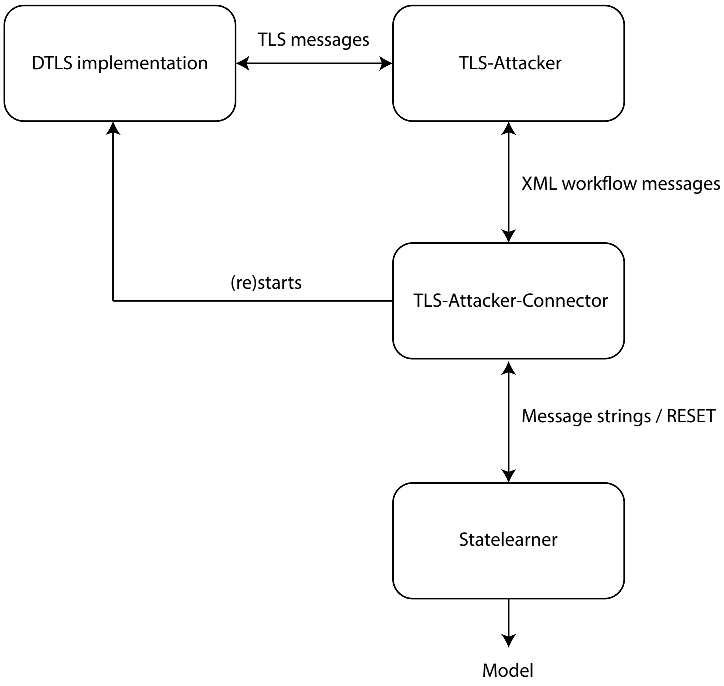

Figure 3.1: Research set-up

Figure 3.1 shows the research set-up we have used to analyse DTLS implementations. The DTLS implementation is communicating with TLS-Attacker. Which message TLS-Attacker should send to the DTLS implementation is being controlled by Statelearner. Statelearner decides what message will be sent. The TLS-Attacker-Connector is translating a string message from Statelearner to a TLS message which is used by TLS-Attacker.

First, we will start the TLS-Attacker-Connector by:

```
$ java -jar target/TLSAttackerConnector2.0.jar -pV DTLS12
-eS OpenSSL
```

TLS-Attacker-Connector will listen to message strings on TCP port 6666. Next, we will start Statelearner by:

```
$ java -jar target/stateLearner-0.0.1-SNAPSHOT.jar
dtls.properties
```

| **Implementation** | **Version** | **URL** |
|---|---|---|
| OpenSSL | 1.1.2-dev | `https://www.openssl.org/` |
| mbedTLS | 2.14.1 | `https://tls.mbed.org/` |
| WolfSSL | 3.15.5 | `https://www.wolfssl.com/` |

Table 3.2: Tested implementations

Table 3.2 shows the tested implementations in this thesis. We discuss each implementation in the next section.

## 3.3 OpenSSL

OpenSSL is an open-source toolkit for TLS [16]. It is written in C and is maintained by a team of committers. In this thesis, we have used the 1.1.2-dev version of OpenSSL. To start an OpenSSL server, execute the following command in the command line:

```
$ openssl s_server -cert cert.pem -key key.pem
-port 4433 -dtls1_2
```

You can use your own certificate and key pair, but you can also use the certificate and key pair created by the `keygen.sh` of TLS-Attacker. These are stored in the `resources/` folder of TLS-Attacker. Starting the TLS-Attacker-Connector with the -eS OpenSSL flag will start an OpenSSL server with the following command:

```
$ openssl s_server -key /TLS-Attacker/resources/rsa1024key.pem
cert /TLS-Attacker/resources/rsa1024cert.pem -dtls1_2
```

```
-state -timeout 1 -mtu 1500
```

The flags means the following:

- -key: Private key
- -cert: Certificate file
- -dtls1_2: Use DTLS 1.2
- -state: Prints TLS state of the server
- -timeout: Sets timeout to 1 second
- -mtu: Sets the maximum transmission unit to 1500. By default, the link layer maximum transmission unit is 1500 bytes. However, whithout this -mtu flag OpenSSL does not work in our case.

Appendix A.1 and Appendix A.2 show the learned state machine of an OpenSSL DTLS server. Appendix A.1 is transformed into a more readable state machine. This has been done by merging state transitions with the same begin state and end state. The happy path of the handshake has been denoted by green. Appendix A.2 is the original state machine learned by Statelearner. Table 3.3 shows the log file of Statelearner:

| **Name** | **Value** |
|---|---|
| Rounds | 1 |
| Nr. of Membership Queries | 342 |
| Nr. of Equivalence Queries | 100 |
| States in final hypothesis | 8 |
| Time learning | 392 seconds |
| Time searching for counter example | 146 seconds |
| Total time | 538 seconds |

Table 3.3: Statelearner log of OpenSSL

We follow the happy path of a TLS handshake below, denoted by green in Appendix A.1, step by step:

- **State 0**: The handshake starts in $s_0$. No message has been sent yet. From this state each state transition belongs to a handshake message. For example, the state transition $s_0 \rightarrow s_3$. When sending a ChangeCipherSpec message to the OpenSSL server in this state, the server responds with nothing. This is represented with a hyphen $(-)$. Being in state $s_3$, there is no option to get out. So, starting with a

ChangeCipherSpec message, it is not possible to end the conversation with a private connection.

- **State 1**: A ClientHello message has been sent from $s_0$, this results in a state transition from $s_0 \rightarrow s_1$. DTLS sends the ClientHello twice in order to prevent DoS attacks. The server has responded with a HelloVerifyRequest. Again, from this state every message has its own state transition.
- **State 4**: Following the handshake of DTLS in figure 2.3, the ClientHello message has been sent twice when the implementation has reached this state, via $s_0 \rightarrow s_1 \rightarrow s_4$. The server sends ServerHello, Certificate, ServerHelloDone messages. This is expected behaviour from the server.
- **State 5**: The next expected message from the client is a ClientKeyExchange message. In our case, we send an RSAClientKeyExchange. Following the state transitions from the start, $s_0 \rightarrow s_1 \rightarrow s_4 \rightarrow s_5$, we end in state 5. The server did not send any messages, because it is waiting for more messages from our side.
- **State 6**: In this state, we have sent the ChangeCipherSpec. The state transitions thus far is $s_0 \rightarrow s_1 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6$. Still no message from the server during this state transition, which is correct.
- **State 7**: Ending in state $s_7$, our client sent a Finished message. The Finished message is the last message of the handshake and the server answered with a ChangeCipherSpec and Finished message. The handshake is now completed and the server and client can communicate privately. The state transitions from the start are: $s_0 \rightarrow s_1 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7$. This is the happy path of the handshake. Client and server are now able to send application data over the network.

The OpenSSL implementation of DTLS 1.2 would be insecure if there was another path between $s_0$ and $s_7$. We could easily check if there is a second path, by reversing from $s_7$ to $s_0$.

- **State 7**:
  - $s_6 \rightarrow s_7$:
    This is the happy path of the handshake. This is normal behaviour.
  - $s_7 \rightarrow s_7$:
    This is a loop in $s_7$, this is normal behaviour too. From the server's perspective this would be a retransmission by the client. It could be that the message from the server did not reach our client. Thus our client would retransmit its last sent message.

- **State 6**:
  - $s_5 \rightarrow s_6$:
    This is the happy path of the handshake. This is normal behaviour.
  - $s_6 \rightarrow s_6$:
    There a two loops in $s_6$. A loop where the ApplicationData message has been sent and a loop where AlertWarningCloseNotify has been sent. In both cases, the server did not send a message back.
- **State 5**:
  - $s_4 \rightarrow s_5$:
    This is the happy path of the handshake. This is normal behaviour.
- **State 4**:
  - $s_1 \rightarrow s_4$:
    This is the happy path of the handshake. This is normal behaviour.
- **State 1**:
  - $s_0 \rightarrow s_1$:
    This is the happy path of the handshake. This is normal behaviour.
- **State 0**:
  This is the begin state, there is no incoming state transition.

We did not found any other path from $s_0$ to $s_7$, so we have found **no vulnerabilities in OpenSSL**.

## 3.4 mbedTLS

mbedTLS is previously known as PolarSSL [17]. mbedTLS advertises itself as easy to use and easy to get. We have used the 2.14.1 version of mbedTLS. After compilation of the program, an example DTLS server is available in the `programs/ssl` directory. We will use this DTLS server and we have not changed anything in the code base. The example server starts a DTLS server on port 4433. To execute this, use the following command:

```
$ ./dtls_server
```

Starting the TLS-Attacker-Connector with the -eS mbedTLS flag will start a mbedTLS server with the same command:

```
$ /mbedtls-2.14.1/programs/ssl/dtls_server
```

Appendix A.3 and Appendix A.4 show the learned state machine of mbedTLS. As we did for OpenSSL, Appendix A.3 is transformed into a more readable state machine. Table 3.4 shows the log file of Statelearner:

| **Name** | **Value** |
|---|---|
| Rounds | 2 |
| Nr. of Membership Queries | 631 |
| Nr. of Equivalence Queries | 156 |
| States in final hypothesis | 6 |
| Time learning | 874 seconds |
| Time searching for counter example | 217 seconds |
| Total time | 1091 seconds |

Table 3.4: Statelearner log of mbedTLS

Again, we will follow the happy path step by step:

- **State 0**:
  This is the begin state of the implementation.
- **State 1**:
  Sending the first ClientHello results in the state transition $s_0 \rightarrow s_1$. The server responded with a HelloVerifyRequest.
- **State 3**:
  Sending the second ClientHello results in the state transition $s_0 \rightarrow s_1 \rightarrow s_3$. The server responded with the three ServerHello, Certificate, serverHelloDone messages. All normal behaviour thus far.
- **State 4**:
  Sending the RSAClientKeyExchange message results in the state transition $s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_4$. The server did not respond with a message, because it is waiting for the next messages to come.
- **State 5**:
  Sending the ChangeCipherSpec message results in the state transition $s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5$.
- **State 2**:
  Sending the final Finished message, results in a ALERT_FATAL_DE-

CODE_ERROR. The server was not able to decrypt the encrypted finished message. and send a FATAL ERROR. The handshake has failed.

Running Statelearner, ends with a failed handshake. The server was unable to decrypt the Finished message. We were not able to find the reason for the failed handshake. The number of packets sent during the analysis, according to WireShark, was 23756. We expect that a retransmission has taken place during the handshake, but we did not verify this. The cause of this error could be a wrong hash of all previous handshake messages in the Finished message. As explained in the basics of TLS 1.2, a verify data value is computed. To compute this value, we need a hash of all previous handshake messages. If a message has been changed, by for example a retransmission, the hash changes and the verify data has a different value. This will fail the handshake.

Running mbedTLS and TLS-Attacker-Connector without Statelearner would give more insight in what happened. Starting mbedTLS and TLS-Attacker-Connector with the - -test flag, results in a complete handshake and sending application data in both directions. However, after the handshake and application data, mbedTLS fails on processing the clientHello handshake message. Following the communication with Wireshark, we can see the complete handshake and application data, but we cannot see a ClientHello at the end.

The problems seem to point towards a different problem in TLS-Attacker. Future work has to be done on TLS-Attacker in order to analyse a fullhandshake with mbedTLS. We can analyse whether mbedTLS has vulnerabilities if the handshake succeeds and application data can be sent.

As we did with OpenSSL, we can reverse the happy path from $s_5$ to $s_0$ to find any weird behaviour of mbedTLS:

- **General remark**: In state $s_0, s_1, s_4$ and $s_5$ of the happy path, we can send application data to the server without a state change.
- **State 5**:
  - $s_4 \rightarrow s_5$:
    This is the happy path of the handshake. This is normal behaviour.
- **State 4**:
  - $s_3 \rightarrow s_4$:
    This is the happy path of the handshake. This is normal behaviour.
- **State 3**:

  - $s_1 \rightarrow s_3$:
    This is the happy path of the handshake. This is normal behaviour.

- **State 1**:
  - $s_0 \rightarrow s_1$:
    This is the happy path of the handshake. This is normal behaviour.
  - $s_3 \rightarrow s_1$:
    We are able to take a step back from $s_3$ to $s_1$. Sending three ClientHello messages is equal to sending one ClientHello message. This is weird behaviour.
  - $s_4 \rightarrow s_1$:
    Sending a ClientHello, RSAClientKeyExchange or Finished message from $s_4$ results in a ALERT_FATAL_DECODE_ERROR. However, we end up in $s_1$, where we can continue our handshake with a ClientHello message. This is weird behaviour.
  - $s_1 \rightarrow s_1$:
    Sending the messages RSAClientKeyExchange, Finished or AlertWarningCloseNotify, the implementation stays in the same state. This is weird behaviour.

- **State 0**:
  - $s_1 \rightarrow s_0$:
    This is the happy path of the handshake. This is normal behaviour.
  - $s_1 \rightarrow s_1$:
    Sending the messages RSAClientKeyExchange, Finished or AlertWarningCloseNotify, the implementation stays in the same state. This is weird behaviour.

We were able to send an RSAClientKeyExchange or a Finished message before and during the ClientHello messages phase of the handshake without receiving a FATAL_ERROR. This is unexpected behaviour, because the message flow of a TLS handshake is strict. A TLS server must terminate a connection whenever an out of order message is received.

It was also possible to send three ClientHello messages, which ends in the same state as when one ClientHello message was sent. Starting a handshake with four ClientHello messages and continue the handshake is possible with mbedTLS. Increasing this value would not necessarily result in a DoS possibility as creating a HelloVerifyRequest is not a difficult operation. A HelloVerifyRequest message is not a large, an amplification attack would not work either.

Later in the handshake, after the RSAClientKeyExchange, it is possible to send either a ClientHello, RSAClientKeyExchange or Finished Message. This ends in the same state when a normal handshake would have sent one ClientHello message. Since we were not able to perform a successful handshake, we can not state that this is a vulnerability in mbedTLS. However, this could lead to a vulnerability. We have found **three weird behaviours in mbedTLS**.

## 3.5 WolfSSL

WolfSSL is also an open-source library, written in C [18]. WolFSSL is a lightweight, portable TLS library. It targets on IoT and embedded systems, because of its small size and high speed. WolfSSL supports up to TLS 1.3 and DTLS 1.2. In this thesis we use WolfSSL version 3.15.5. After downloading WolfSSL, you have to configure WolfSSL, to enable DTLS by `$ ./configure --enable-dtls --enable-debug`. We also enable more debug information. Then install WolfSSL by `$ make` and `$ sudo make install`.

To start a TLS server example, you need to execute the following command:

```
$ examples/server/server -k /TLS-Attacker-Development/
resources/rsa1024key.pem -c /TLS-Attacker-Development/resources/
rsa1024cert.pem -p 4433
```

Again, we use the key and certificate generated by TLS-Attacker. To start a DTLS server, add the `-u` flag.

Testing the WolfSSL DTLS server with TLS-Attacker-Connector results in an error -373 and error message “No ClientHello before ClientKeyExchange”. Error -373 means that a received message is out of order [19]. Viewing the handshake via Wireshark, we have noticed a weird message flow. The client sends a ClientHello twice. Then, the server answers with HelloVerifyRequest twice. After that the client responds with a ClientKeyExchange. We expect that the TLS-Attacker sends the second ClientHello too fast. Increasing the timeout in TLS-Attacker from 100, to any higher value did not work.

No model has been learned for WolfSSL. In order to learn a model, work has to be done on TLS-Attacker in order to complete a full handshake. If a full handshake succeeds and application data can be send, we can analyse whether WolfSSL has vulnerabilities.

## 3.6 Implementation fingerprinting

As OpenSSL and mbedTLS behave differently in our research, it is possible to distinguish these two. Both implementations have a unique fingerprint when it comes to reacting to certain message flows. Sending a message flow of ClientHello, RSAClientKeyExchange, ClientHello, RSAClientKeyExchange, ChangeCipherSpec and Finished the mbedTLS implementation behaves differently than OpenSSL. MbedTLS allows this behaviour and continues the handshake while OpenSSL terminates the connection. The different behaviour could lead to an attack.

An attacker could use this information to identify which DTLS server is being used. If an application is using and one of them and it has a vulnerability, the attacker could use this information in the Reconnaissance phase of its attack [20]. In this phase, the attacker searches for vulnerabilities in the target. With this information, the attacker could proceed to phase two: the Infiltration phase.

# Chapter 4

# Related Work

Research in the area of state machine inference has been done already. In 2015, 9 different implementations of TLS have been analysed [21]. GnuTLS, mbedTLS, miTLS and OpenSSL were among the 9 implementations. In 3 out of 9 implementations they have found security vulnerabilities. Another paper analysed 145 versions of OpenSSL and LibreSSL both client- and server-side [22]. This paper found 15 unique OpenSSL and 2 LibreSSL server-side state machines and found 9 unique OpenSSL and 2 LibreSSL client-side state machines. In 2018, TLS 1.3 was analysed using state machine inference [23]. This thesis found unexpected error alert messages on some inputs in WolfSSL.

Next to the TLS protocol, also the Secure Shell (SSH) protocol has been analysed with state machine inference [24] [25] [26]. SSH is a protocol that provides a secure remote login over an insecure network [27].

DTLS implementations have not been analysed with state machine inference. However, DTLS has been analysed to implement Constrained Application Protocol (CoAP) in IoTs [28]. CoAP is a protocol that provides interaction between applications in a constrained environment [29]. This paper concludes that there is an issue of usability with deploying DTLS with constrained devices in IoT. The paper suggests new lightweight security mechanisms to secure CoAP.

The record protocol of TLS and DTLS have been attacked [30]. Vulnerabilities have been found in the record protocol of TLS and DTLS when used in CBC-mode.

This thesis analyses DTLS implementations by using state machine inference, which has not been done before.

# Chapter 5

# Conclusions

In this thesis, we were able to create a state machine of OpenSSL and mbedTLS. For WolfSSL, we were not able to create a state machine. During the handshake with WolfSSL, we received an out of order error. No vulnerabilities has been found in the server-side DTLS implementation of OpenSSL. We were able to perform a handshake and send application data to the OpenSSL server. Using any message flow other than the normal handshake, it was not possible to perform a successful handshake and to send application data.

With mbedTLS, the handshake failed during the final Finished message. The mbedTLS server was not able to decode the Finished message and sent an ALERT FATAL DECODE ERROR message back. We expect that a retransmission has been taken place and TLS-Attacker is not able to handle retransmission. We can not confirm this as 23756 packets has been sent during the creation of the state machine. However, we have found three unexpected behaviours earlier in the handshake.

WolfSSL received messages out of order. TLS-Attacker sends the handshake messages in the same order as for OpenSSL and mbedTLS. Increasing the timeout of TLS-Attacker did not solve the problem.

As mbedTLS and OpenSSL behaved differently during our research, it is possible to distinguish these two. Both implementations have a unique fingerprint when it comes to reacting to certain message flows. This will help an attacker to identify a target in the reconnaissance phase of launching an attack.

Future work has to be done on the software tool TLS-Attacker in order to fix the problems with mbedTLS and WolfSSL. If TLS-Attacker can successfully complete a handshake and send application data to mbedTLS and WolfSSL, it is possible to further analyse whether there are vulnerabilities in these implementations. As we were not able to perform a successful handshake, we can not state that the unexpected behaviour of mbedTLS results in a vulnerability. A state machine including a successful handshake with mbedTLS and WolfSSL could reveal more unexpected behaviour or vulnerabilities.

# Bibliography

[1] University of Southern California. Transmission control protocol, 1981. [RFC793] `https://tools.ietf.org/html/rfc793`.

[2] J. Postel. User datagram protocol, 1980. [RFC768] `https://tools.ietf.org/html/rfc768`.

[3] Min Zhang, Maurizio Dusi, Wolfgang John, and Changjia Chen. Analysis of UDP traffic usage on internet backbone links. In *Applications and the Internet, 2009. SAINT'09. Ninth Annual International Symposium on*, pages 280–281. IEEE, 2009.

[4] Dierks, T. and E. Rescorla. The transport layer security (TLS) protocol version 1.2, 2008. [RFC5246] `https://tools.ietf.org/html/rfc5246`.

[5] Rescorla, E. and Modadugu, N. Datagram transport layer security, 2006. [RFC4347] `https://tools.ietf.org/html/rfc4347`.

[6] Rescorla, E. and Modadugu, N. Datagram transport layer security version 1.2, 2012. [RFC6347] `https://tools.ietf.org/html/rfc6347`.

[7] Tschofenig, H. Rescorla, E. and Modadugu, N. The datagram transport layer security (DTLS) protocol version 1.3 draft-ietf-tls-dtls13-30, 2018. [draft-ietf-tls-dtls13-30] `https://tools.ietf.org/html/draft-ietf-tls-dtls13-30`.

[8] Dierks, T. and C. Allen. The TLS protocol version 1.0, 1999. [RFC2246] `https://tools.ietf.org/html/rfc2246`.

[9] Dierks, T. and E. Rescorla. The transport layer security (TLS) protocol version 1.1, 2006. [RFC4346] `https://tools.ietf.org/html/rfc4346`.

[10] E. Rescorla. The transport layer security (TLS) protocol version 1.3, 2018. [RFC8446] `https://tools.ietf.org/html/rfc8446`.

[11] George H Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[12] Ruhr University Bochum and Hackmanit GmbH. Tls-attacker. `https://github.com/RUB-NDS/TLS-Attacker`. Accessed: 2019-01-15.

[13] Tls-attacker-connector. `https://gitlab.science.ru.nl/joeri/TLS-Attacker-Connector`. Accessed: 2019-01-15.

[14] Statelearner. `https://github.com/jderuiter/statelearner`. Accessed: 2019-01-15.

[15] Learnlib. `https://github.com/LearnLib/learnlib`.

[16] OpenSSL version 1.1.2-dev. `https://github.com/openssl/openssl`. Accessed: 2018-10-28.

[17] mbedTLS version 2.14.1. `https://tls.mbed.org/download/start/mbedtls-2.14.1-apache.tgz`. Accessed: 2018-11-02.

[18] WolfSSL version 3.15.5. `https://github.com/wolfSSL/wolfssl`. Accessed: 2018-11-08.

[19] WolfSSL error list. `https://github.com/wolfSSL/wolfssl/blob/master/wolfssl/error-ssl.h`. Accessed: 2019-01-15.

[20] HP Sanghvi and MS Dahiya. Cyber reconnaissance: an alarm before cyber attack. *International Journal of Computer Applications*, 63(6), 2013.

[21] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security Symposium*, pages 193–206, 2015.

[22] Joeri de Ruiter. A tale of the OpenSSL state machine: A large-scale black-box analysis. In *Nordic Conference on Secure IT Systems*, pages 169–184. Springer, 2016.

[23] Jules van Thoor, Joeri de Ruiter, and Erik Poll. Learning state machines of TLS 1.3 implementations, 2018. Bachelor Thesis. Radboud University.

[24] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 142–151. ACM, 2017.

[25] Toon Lenaerts, Frits Vaandrager, Paul Fiterău-Broştean, and Erik Poll. Improving-protocol state fuzzing of SSH, 2018. Bachelor Thesis. Radboud University.

[26] Patrick Verleg, Erik Poll, and Frits Vaandrager. Inferring SSH state machines using protocol state fuzzing, 2016. Master's Thesis. Radboud University.

[27] Ylonen, T. and Lonvick, C. The secure shell (SSH) transport layer protocol, 2006. [RFC4253] `https://tools.ietf.org/html/rfc4253`.

[28] Thamer A Alghamdi, Aboubaker Lasebae, and Mahdi Aiash. Security analysis of the constrained application protocol in the internet of things. In *Future Generation Communication Technology (FGCT), 2013 second international conference on*, pages 163–168. IEEE, 2013.

[29] Hartke, K. Shelby, Z. and Bormann, C. The constrained application protocol (CoAP), 2014. [RFC7252] `https://tools.ietf.org/html/rfc7252`.

[30] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540. IEEE, 2013.

## Appendix A

# Appendix

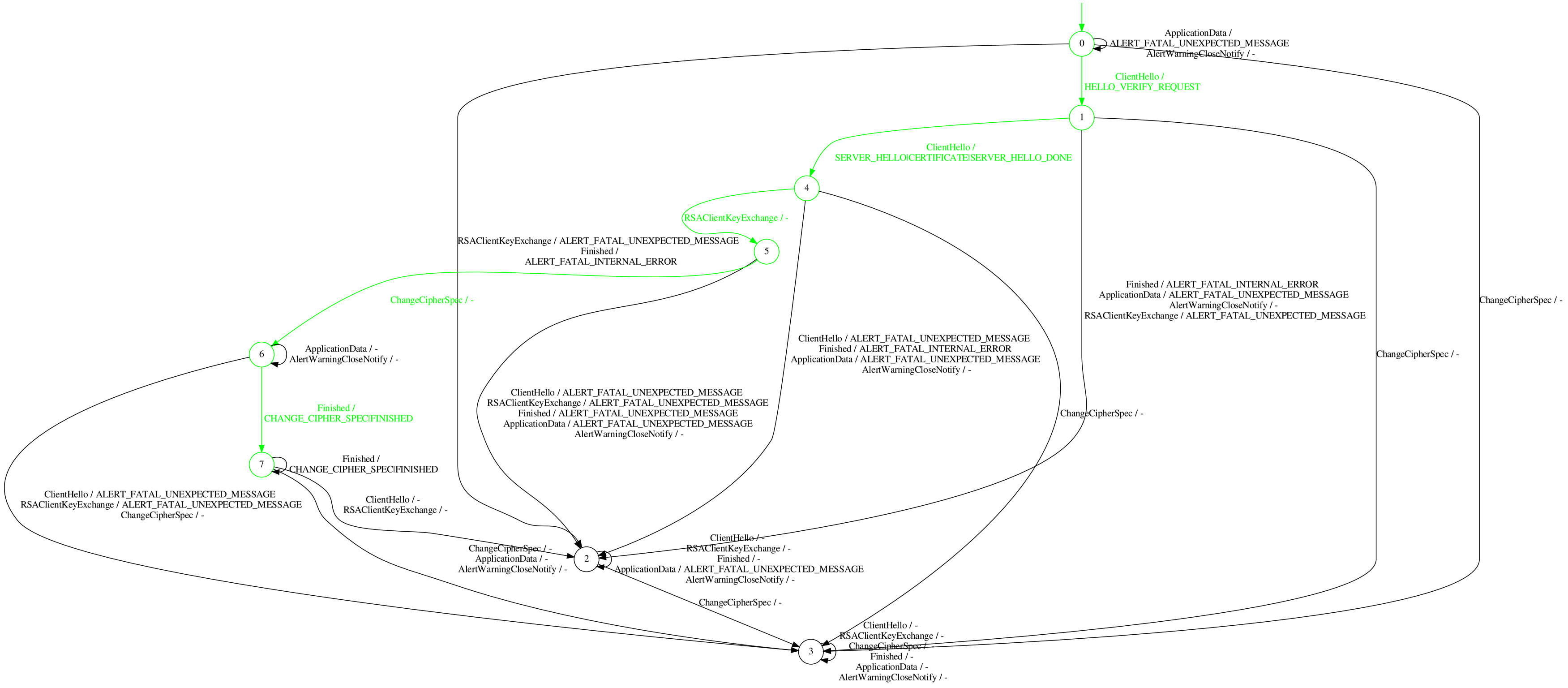
0
ApplicationData / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
ClientHello / HELLO_VERIFY_REQUEST
1
ClientHello / SERVER_HELLO|CERTIFICATE|SERVER_HELLO_DONE
4
RSAClientKeyExchange / -
5
RSAClientKeyExchange / ALERT_FATAL_UNEXPECTED_MESSAGE
Finished / ALERT_FATAL_INTERNAL_ERROR
ChangeCipherSpec / -
6
ApplicationData / -
AlertWarningCloseNotify / -
Finished / CHANGE_CIPHER_SPEC|FINISHED
7
Finished / CHANGE_CIPHER_SPEC|FINISHED
ClientHello / -
RSAClientKeyExchange / -
ClientHello / ALERT_FATAL_UNEXPECTED_MESSAGE
RSAClientKeyExchange / ALERT_FATAL_UNEXPECTED_MESSAGE
ChangeCipherSpec / -
ClientHello / ALERT_FATAL_UNEXPECTED_MESSAGE
RSAClientKeyExchange / ALERT_FATAL_UNEXPECTED_MESSAGE
Finished / ALERT_FATAL_UNEXPECTED_MESSAGE
ApplicationData / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
ClientHello / ALERT_FATAL_UNEXPECTED_MESSAGE
Finished / ALERT_FATAL_INTERNAL_ERROR
ApplicationData / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
Finished / ALERT_FATAL_INTERNAL_ERROR
ApplicationData / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
RSAClientKeyExchange / ALERT_FATAL_UNEXPECTED_MESSAGE
ChangeCipherSpec / -
ChangeCipherSpec / -
ChangeCipherSpec / -
ChangeCipherSpec / -
ApplicationData / -
AlertWarningCloseNotify / -
2
ClientHello / -
RSAClientKeyExchange / -
Finished / -
ApplicationData / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
ChangeCipherSpec / -
3
ClientHello / -
RSAClientKeyExchange / -
ChangeCipherSpec / -
Finished / -
ApplicationData / -
AlertWarningCloseNotify / -

Figure A.1: OpenSSL DTLS server state machine (Reformatted)

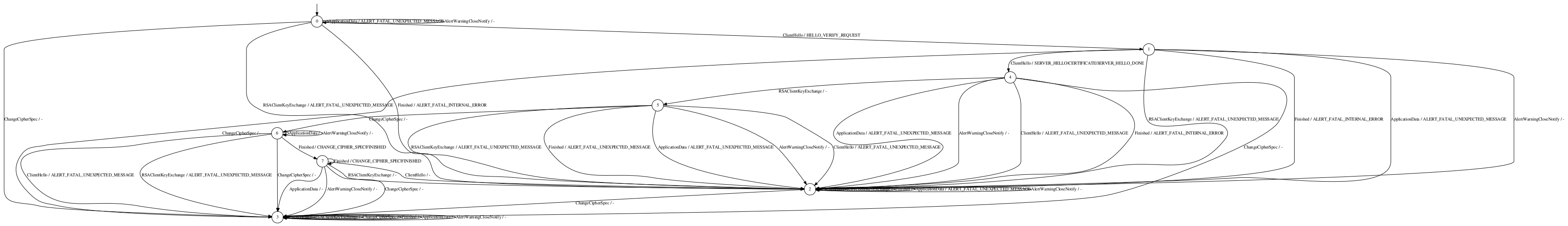
0
1
2
3
4
5
6
7
ApplicationData / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
ClientHello / HELLO_VERIFY_REQUEST
ClientHello / SERVER_HELLO|CERTIFICATE|SERVER_HELLO_DONE
RSAClientKeyExchange / ALERT_FATAL_UNEXPECTED_MESSAGE
Finished / ALERT_FATAL_INTERNAL_ERROR
ChangeCipherSpec / -
ApplicationData / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
RSAClientKeyExchange / -
ApplicationData / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
ClientHello / ALERT_FATAL_UNEXPECTED_MESSAGE
Finished / ALERT_FATAL_INTERNAL_ERROR
ChangeCipherSpec / -
RSAClientKeyExchange / ALERT_FATAL_UNEXPECTED_MESSAGE
Finished / ALERT_FATAL_INTERNAL_ERROR
ChangeCipherSpec / -
RSAClientKeyExchange / ALERT_FATAL_UNEXPECTED_MESSAGE
Finished / ALERT_FATAL_UNEXPECTED_MESSAGE
ApplicationData / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
ClientHello / ALERT_FATAL_UNEXPECTED_MESSAGE
ChangeCipherSpec / -
ApplicationData / -
AlertWarningCloseNotify / -
Finished / CHANGE_CIPHER_SPECIFINISHED
ChangeCipherSpec / -
ClientHello / ALERT_FATAL_UNEXPECTED_MESSAGE
RSAClientKeyExchange / ALERT_FATAL_UNEXPECTED_MESSAGE
ChangeCipherSpec / -
Finished / CHANGE_CIPHER_SPECIFINISHED
RSAClientKeyExchange / -
ClientHello / -
ApplicationData / -
AlertWarningCloseNotify / -
ChangeCipherSpec / -
ApplicationData / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
ChangeCipherSpec / -
ClientHello / -
RSAClientKeyExchange / -
ChangeCipherSpec / -
Finished / -
ApplicationData / -
AlertWarningCloseNotify / -

Figure A.2: OpenSSL DTLS server state machine

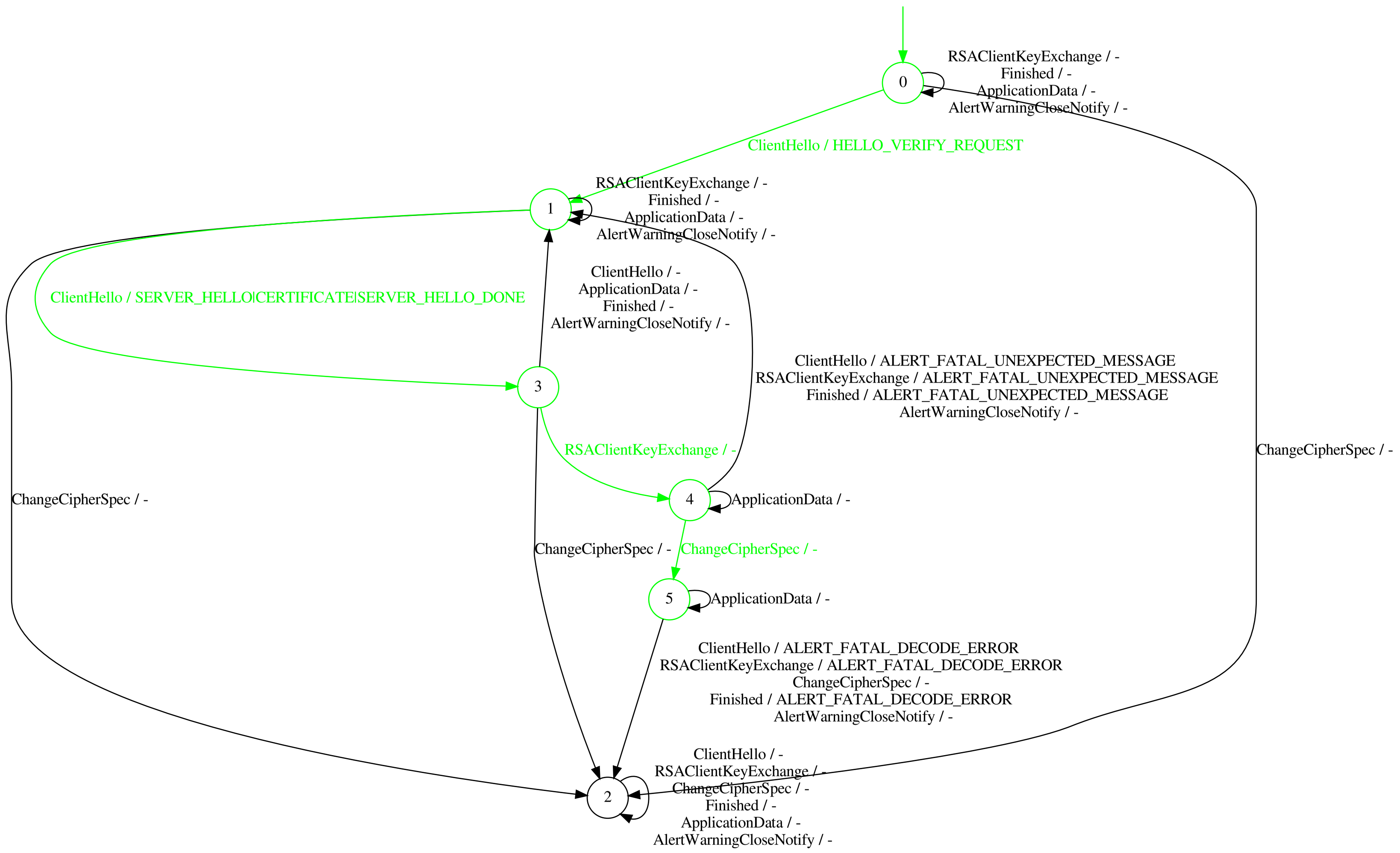
0
RSAClientKeyExchange / -
Finished / -
ApplicationData / -
AlertWarningCloseNotify / -
ClientHello / HELLO_VERIFY_REQUEST
1
RSAClientKeyExchange / -
Finished / -
ApplicationData / -
AlertWarningCloseNotify / -
ClientHello / SERVER_HELLO|CERTIFICATE|SERVER_HELLO_DONE
ClientHello / -
ApplicationData / -
Finished / -
AlertWarningCloseNotify / -
3
ClientHello / ALERT_FATAL_UNEXPECTED_MESSAGE
RSAClientKeyExchange / ALERT_FATAL_UNEXPECTED_MESSAGE
Finished / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
RSAClientKeyExchange / -
ChangeCipherSpec / -
ChangeCipherSpec / -
4
ApplicationData / -
ChangeCipherSpec / -
ChangeCipherSpec / -
5
ApplicationData / -
ClientHello / ALERT_FATAL_DECODE_ERROR
RSAClientKeyExchange / ALERT_FATAL_DECODE_ERROR
ChangeCipherSpec / -
Finished / ALERT_FATAL_DECODE_ERROR
AlertWarningCloseNotify / -
2
ClientHello / -
RSAClientKeyExchange / -
ChangeCipherSpec / -
Finished / -
ApplicationData / -
AlertWarningCloseNotify / -

Figure A.3: mbedTLS DTLS server state machine (Reformatted)

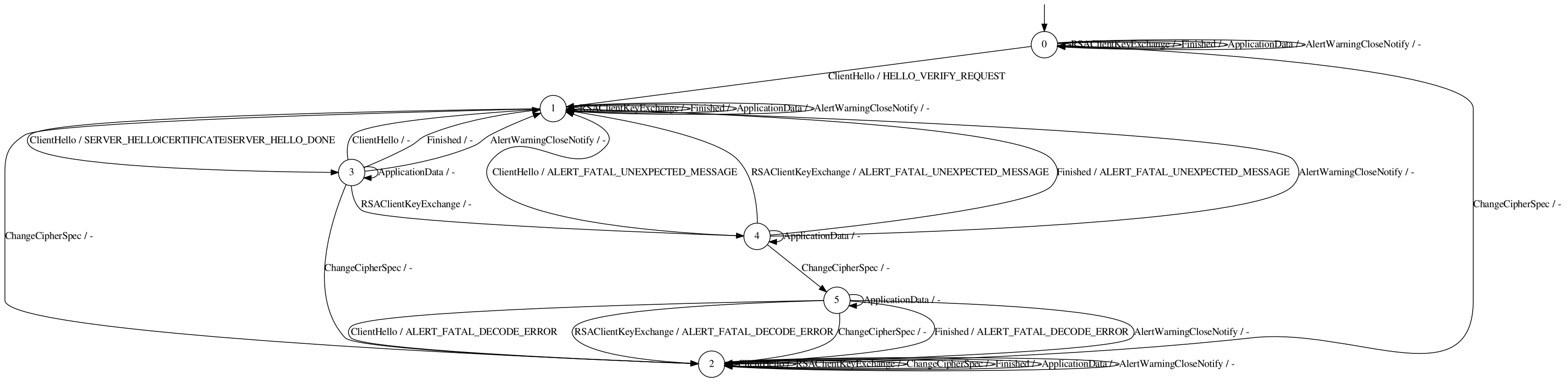
0
RSAClientKeyExchange / - Finished / - ApplicationData / - AlertWarningCloseNotify / -
ClientHello / HELLO_VERIFY_REQUEST
1
RSAClientKeyExchange / - Finished / - ApplicationData / - AlertWarningCloseNotify / -
ClientHello / SERVER_HELLO|CERTIFICATE|SERVER_HELLO_DONE
ClientHello / -
Finished / -
AlertWarningCloseNotify / -
3
ApplicationData / -
ClientHello / ALERT_FATAL_UNEXPECTED_MESSAGE
RSAClientKeyExchange / ALERT_FATAL_UNEXPECTED_MESSAGE
Finished / ALERT_FATAL_UNEXPECTED_MESSAGE
AlertWarningCloseNotify / -
RSAClientKeyExchange / -
ChangeCipherSpec / -
4
ApplicationData / -
ChangeCipherSpec / -
ChangeCipherSpec / -
5
ApplicationData / -
ClientHello / ALERT_FATAL_DECODE_ERROR
RSAClientKeyExchange / ALERT_FATAL_DECODE_ERROR
ChangeCipherSpec / -
Finished / ALERT_FATAL_DECODE_ERROR
AlertWarningCloseNotify / -
ChangeCipherSpec / -
2
ClientHello / - RSAClientKeyExchange / - ChangeCipherSpec / - Finished / - ApplicationData / - AlertWarningCloseNotify / -

Figure A.4: mbedTLS DTLS server state machine