

# TLS 1.3 Stack for strongSwan

## Technical Documentation

Bachelor Thesis, Fall Term 2020

Department of Computer Science

University of Applied Sciences Rapperswil (HSR)

[www.ost.ch](http://www.ost.ch)

Author: Pascal Knecht ([pascal.knecht@ost.ch](mailto:pascal.knecht@ost.ch), @ryru\_foo )

Advisors: Prof. Dr. Andreas Steffen; Tobias Brunner, HSR

Project Partner: Institute for Networked Solutions (INS)

External Co-Examiner: Dr. Ralf Hauser, PrivaSphere AG

Internal Co-Examiner: Prof. Frank Koch

Date: 19th December 2020

# Contents

<b>1. Abstract</b>	<b>10</b>
<b>2. Introduction</b>	<b>11</b>
2.1. Overview . . . . .	11
2.2. Transport Layer Security Protocol (TLS) . . . . .	12
2.3. strongSwan . . . . .	16
2.4. Project Scope . . . . .	17
<b>3. Theory</b>	<b>19</b>
3.1. TLS handshakes . . . . .	19
3.2. 1-RTT TLS 1.3 handshake . . . . .	20
3.3. Mutual authentication TLS 1.3 handshake . . . . .	22
3.4. PSK session resumption TLS 1.3 handshake . . . . .	22
3.5. Incorrect DHE Share . . . . .	24
3.6. TLS 1.3 version negotiation . . . . .	25
3.7. HMAC-based Key Derivation Function . . . . .	26
<b>4. Concept</b>	<b>30</b>
4.1. Architecture Overview of libtls . . . . .	30
4.2. Implementation design choice . . . . .	31
4.3. PSK . . . . .	33
4.4. HKDF . . . . .	38
4.5. Key Derivation and Switching . . . . .	41
<b>5. Implementation</b>	<b>44</b>
5.1. Server-side implementation . . . . .	44
5.2. Mutual authentication . . . . .	49
5.3. PSK-based resumption . . . . .	53
5.4. Unit Tests . . . . .	57
<b>6. Testing</b>	<b>60</b>
6.1. Client . . . . .	61
6.2. Server . . . . .	64
6.3. Mutual Authentication . . . . .	72
<b>7. Results</b>	<b>75</b>
7.1. Achievements . . . . .	75
7.2. Further work . . . . .	77
7.3. Outlook to TLS 1.4? . . . . .	78
<b>A. List of Abbreviations</b>	<b>80</b>
<b>B. Bibliography</b>	<b>82</b>

<b>C. Compile Instructions</b>	<b>85</b>
C.1. Compile and Unit Tests . . . . .	85
<b>D. Source Code</b>	<b>89</b>
D.1. Code-Repository . . . . .	89
<b>E. OpenSSL Commands</b>	<b>90</b>
E.1. Handshake TLS 1.3 client to TLS 1.3 server . . . . .	90
E.2. Handshake TLS 1.2 client to TLS 1.3 server . . . . .	90
E.3. Handshake TLS 1.3 client to TLS 1.2 server . . . . .	91
E.4. KeyUpdate messages . . . . .	91
E.5. HelloRetryRequest . . . . .	91
E.6. Mutual authentication . . . . .	92
E.7. PSK session resumption . . . . .	92
<b>F. Key material</b>	<b>94</b>

# List of Figures

2.1.	The location of TLS 1.3 in the four layer TCP/IP reference model defined by the ARPA of the DoD. . . . .	12
2.3.	Full handshake in TLS 1.2 with server authentication. . . . .	13
2.2.	Network protocol stack with the four TLS sub-protocols (yellow). . . . .	13
2.4.	Full handshake in TLS 1.3 with server authentication. . . . .	14
3.1.	Explanation of shape use for messages and extensions. . . . .	20
3.2.	Regular 1-RTT TLS 1.3 handshake. . . . .	21
3.3.	Mutual authentication TLS 1.3 handshake. . . . .	22
3.4.	Initial session to use PSK in the following TLS 1.3 connection. . . . .	23
3.5.	PSK session resumption TLS 1.3 handshake. . . . .	24
3.6.	Incorrect DHE Share. . . . .	25
3.7.	Each HKDF-Extract signals a one-way state transition. . . . .	27
4.1.	Package diagram: strongSwan components that rely on a TLS stack . . . .	30
4.2.	First NewSessionTicket message from the server. . . . .	34
4.3.	Second NewSessionTicket message from the server. . . . .	34
4.4.	ClientHello message with pre_shared_key extension in the second connection. . . . .	34
4.5.	ServerHello message with pre_shared_key extension in the second connection. . . . .	35
4.6.	The whole ServerHello message in PSK mode in the second connection. .	35
4.7.	Both handshakes with all messages in PSK mode. . . . .	35
4.8.	PSK key generation. . . . .	37
4.9.	The state machine for our HKDF implementation with the four phases. . . .	39
4.10.	UML class diagram of the current HKDF implementation. . . . .	39
4.11.	Basic concept of authenticated encryption with associated data (AEAD). . .	40
4.12.	Key material derivation and switching in TLS 1.3. . . . .	42
5.1.	HKDF with new interface for PSK. . . . .	53
5.2.	Unit tests before this bachelor thesis. . . . .	57
5.3.	Unit tests after this bachelor thesis. . . . .	57
5.4.	Unit test coverage of libtls before this thesis (lines/functions/branches). .	58
5.5.	Unit test coverage of libtls after this thesis (lines/functions/branches). .	58
5.6.	Unit test coverage of libtls before this thesis in detail. . . . .	59
5.7.	Unit test coverage of libtls after this thesis in detail. . . . .	59
6.1.	tls_test with TLS 1.3 to OpenSSL server. . . . .	61
6.2.	tls_test with TLS 1.2 to OpenSSL server. . . . .	62
6.3.	OpenSSL to tls_test. . . . .	65
6.4.	OpenSSL 1.2 to tls_test. . . . .	66
6.5.	OpenSSL to tls_test TLS 1.2. . . . .	66

6.6. <code>tls_test</code> to <code>tls_test</code> . . . . .	67
6.7. KeyUpdate with OpenSSL server. . . . .	69
6.8. KeyUpdate with OpenSSL client. . . . .	70
6.9. HRR with OpenSSL server. . . . .	70
6.10. HRR with OpenSSL client. . . . .	71
6.11. HRR with OpenSSL server. . . . .	72
6.12. OpenSSL to <code>tls_test</code> . . . . .	72
6.13. <code>tls_test</code> to OpenSSL. . . . .	73
6.14. <code>tls_test</code> to <code>tls_test</code> . . . . .	74
6.15. Client does not provide certificate. . . . .	74

## List of source codes

1.	Example TLS version branching . . . . .	32
2.	TLS version numbers . . . . .	32
3.	TLS peer state machine states . . . . .	33
4.	TLS 1.3 server-side process state machine. . . . .	45
5.	TLS 1.3 server-side build state machine. . . . .	46
6.	Remove RSA-PSS signature schemes. . . . .	46
7.	Allow RSA-PSS signature schemes for verifying. . . . .	47
8.	Server-side process state machine with KeyUpdate support. . . . .	48
9.	Server-side build state machine with KeyUpdate support. . . . .	48
10.	Server-side HRR retrying function. . . . .	48
11.	Server-side process state machine with mutual authentication support. . . . .	50
12.	Server-side build state machine with mutual authentication support. . . . .	51
13.	Static blob to generate client certificate verify signature. . . . .	52
14.	Client-side process state machine with mutual authentication support. . . . .	52
15.	Client-side build state machine with mutual authentication support. . . . .	53
16.	HKDF interface with <code>resume()</code> and <code>binder()</code> . . . . .	54
17.	HKDF <code>resume()</code> implementation. . . . .	55
18.	HKDF <code>binder()</code> implementation. . . . .	56
19.	Help output of <code>tls_test</code> utile tool. . . . .	60
20.	<code>tls_test</code> connection to OpenSSL using TLS 1.3. . . . .	61
21.	<code>tls_test</code> limited to TLS 1.2 connection to OpenSSL. . . . .	62
22.	TLS connection to Google on IP v4 and port 443/tcp using <code>tls_test</code> . . . . .	62
23.	TLS connection to Google on IP v6 and port 443/tcp using <code>tls_test</code> . . . . .	63
24.	TLS connection to <code>tls13.1d.pw</code> on port 443/tcp using <code>tls_test</code> . . . . .	64
25.	OpenSSL connection to <code>tls_test</code> using TLS 1.3. . . . .	65
26.	OpenSSL limited to TLS 1.2 connection to <code>tls_test</code> . . . . .	65
27.	OpenSSL connection to <code>tls_test</code> limited TLS 1.2. . . . .	66
28.	<code>tls_test</code> connection to <code>tls_test</code> using TLS 1.3 client side. . . . .	67
29.	<code>tls_test</code> connection to <code>tls_test</code> using TLS 1.3 server side. . . . .	67
30.	Key update message and key update request from the OpenSSL server. . . . .	68
31.	Key update message and key update request from the OpenSSL client. . . . .	69
32.	HRR message by OpenSSL server. . . . .	70
33.	HRR message by <code>tls_test</code> server. . . . .	71
34.	Connection termination by server due to missing supported groups in client hello. . . . .	71
35.	Mutual authentication with OpenSSL to <code>tls_test</code> . . . . .	72
36.	Mutual authentication with <code>tls_test</code> to OpenSSL. . . . .	73
37.	Mutual authentication with <code>tls_test</code> to <code>tls_test</code> client side. . . . .	73
38.	Mutual authentication with <code>tls_test</code> to <code>tls_test</code> server side. . . . .	74
39.	Client does not provide certificate and server terminates connection. . . . .	74

40.	Installation of required packages. . . . .	85
41.	Building steps to compile the project. . . . .	86
42.	Generate code coverage report. . . . .	88
43.	OpenSSL version used in this chapter. . . . .	90
44.	TLS 1.3 server. . . . .	90
45.	TLS 1.3 client. . . . .	90
46.	TLS 1.3 server. . . . .	90
47.	TLS 1.2 client. . . . .	90
48.	TLS 1.2 server. . . . .	91
49.	TLS 1.3 client. . . . .	91
50.	TLS 1.3 server expects <i>secp521r1</i> key share DH group. . . . .	91
51.	TLS 1.3 client by default sends <i>x25519</i> key share DH group. . . . .	91
52.	TLS 1.3 server expects client to authenticate itself. . . . .	92
53.	TLS 1.3 client authenticates itself with the key material. . . . .	92
54.	Starts an OpenSSL server instance. . . . .	92
55.	Starts an OpenSSL client instance and writes session ticket into client.session file. . . . .	92
56.	Starts an OpenSSL client instance and reads session ticket from client.session file. . . . .	92
57.	Content of the session ticket in file client.session. . . . .	93
58.	Key generation using strongSwan's <i>pki</i> tool. . . . .	94
59.	RSA private key used during this bachelor thesis. . . . .	94
60.	Corresponding X.509 certificate. . . . .	95

## Bachelor Thesis 2020

### TLS 1.3 for strongSwan

Student: Pascal Knecht

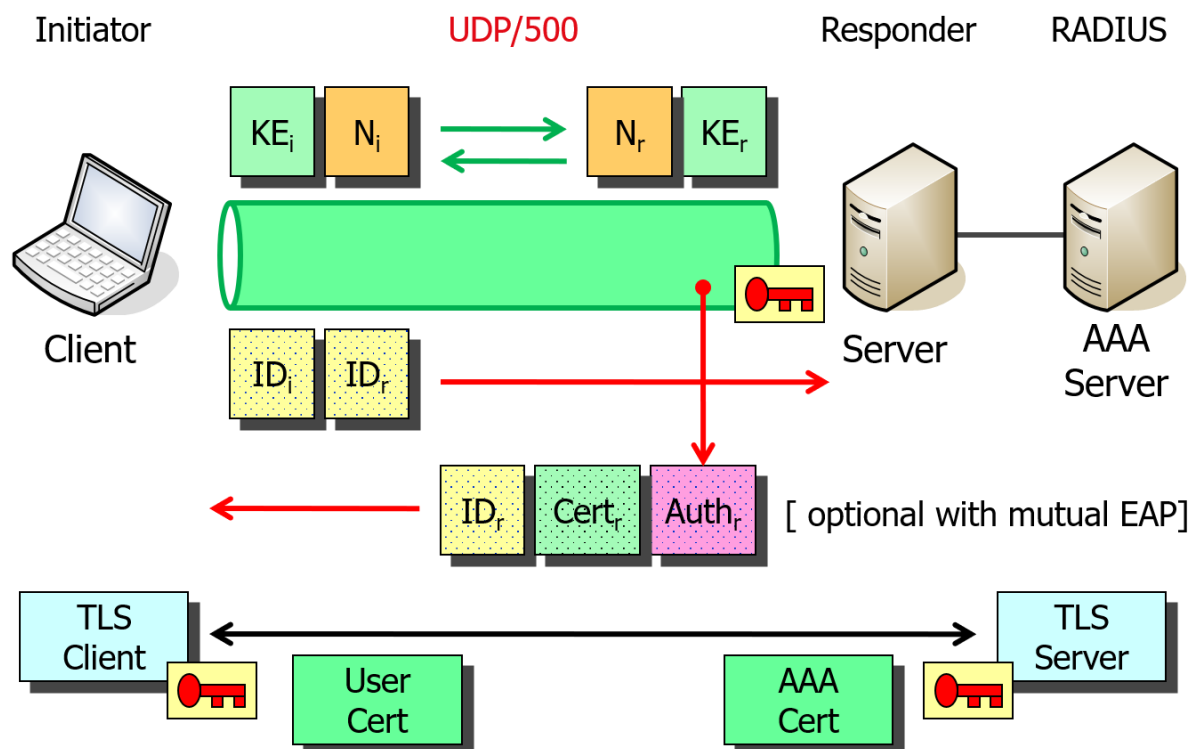
Advisors: Prof. Dr. Andreas Steffen / Tobias Brunner

Issue Date: Monday, August 31<sup>st</sup> 2020

Submission Date: Sunday, December 20<sup>th</sup> 2020

#### Introduction

The strongSwan open source VPN solution has a Transport Layer Security (TLS) stack of its own implemented by the strongSwan `libtls` library [1]. TLS is used by the strongSwan IKEv2 daemon for EAP-based authentication (EAP-TLS, EAP-TTLS, EAP-PEAP) as shown in the figure below but `libtls` could potentially be employed by any stand-alone TLS application.



Due to several deficiencies the TLS 1.2 version supported by strongSwan has been deprecated and therefore an upgrade to the latest TLS 1.3 standard defined by RFC 8446 [2] is urgently required.

In a preliminary semester project the strongSwan `libtls` library was extended with a minimal viable TLS 1.3 client without certificate-based client-side authentication. It is the goal of this bachelor thesis to enhance the strongSwan TLS 1.3 stack with the following features:



## Objectives

### Mandatory:

- Implementation of the TLS 1.3 server-side protocol.
- Implementation of TLS 1.3 client authentication based on X.509 client certificates.
- Interoperability testing of client- and server-side TLS 1.3 strongSwan libtls stack with third party implementations (e.g. OpenSSL, Google).

### Optional:

- PSK-based TLS 1.3 session resumption (0-RTT).
- Evaluation of EAP-TLS/EAP-TTLS/EAP-PEAP interoperability e.g. with an TLS 1.3-enhanced FreeRADIUS server.

## Links

- [1] strongSwan libtls library, github source code repository  
<https://github.com/strongswan/strongswan/tree/master/src/libtls>
- [2] RFC 8446 “The Transport Layer Security (TLS) Protocol Version 1.3”, August 2018  
<https://tools.ietf.org/html/rfc8446>

Rapperswil, August 31<sup>st</sup> 2020



Prof. Dr. Andreas Steffen

# 1. Abstract

**Introduction:** The Transport Layer Security (TLS) protocol secures network connections between a client and a server. It encrypts and authenticates data from higher-level protocols such as the Hypertext Transfer Protocol (HTTP), and guarantees that the information transmitted remains confidential and keeps its integrity. The most widely used TLS version today still is version 1.2 released in 2008 (RFC 5246), even though 1.3 was released in 2018 (RFC 8446). The strongSwan project maintained by the University of Applied Sciences Rapperswil (HSR) is an open-source IPsec implementation written in C. strongSwan features its own TLS stack encapsulated in the library `libtls`. It enables communication-authentication via various EAP authentication methods (TLS, TTLS, PEAP) used to establish an IKEv2 connection. A client-side TLS 1.3 prototype stack was implemented in the preliminary work before this thesis. However, `libtls` does not yet fully support TLS 1.3 in the sense strongSwan requires.

**Objective:** The goal of this bachelor thesis is to implement the TLS 1.3 server-side protocol stack, add support for mutual authentication to enable client authentication in a TLS 1.3 handshake and lastly add support for PSK-based session resumption with TLS 1.3. The former two tasks are mandatory features to make the new TLS 1.3 implementation useful for the strongSwan project and the latter is an optional feature. To achieve these goals, it is necessary to integrate new or adapt existing messages that are exchanged between client and server. In addition, TLS 1.3 requires fundamental changes to the cryptographic mechanisms that enable a secure and authenticated encryption.

Until a connection is established, the handshake passes through various states in a state machine. The state machine has considerably changed in the new version, which also implies that the handshake flow and state machine must be adapted. Moreover, the way cryptographic keys are generated and derived by each peer has fundamentally changed, this also needs to be addressed in this work.

**Results:** TLS 1.3 was successfully implemented and provides a server-side stack and mutual authentication. The TLS 1.3 client-side stack, which was implemented already in the preliminary study term project, was improved significantly. Additionally, smaller but important features such as support for KeyUpdate or HelloRetryRequest messages were implemented. The client and server implementations have been extensively tested against each other and also with external servers and tools such as OpenSSL. The mandatory goals were achieved. The optional goal, the PSK-based session resumption, was not implemented fully due to time constraints but the foundation has been laid: The cryptographic logic encapsulated in the HKDF implementation is able to provide all the necessary secrets. However, the communication protocol and logic implementation remains open to further work. Nevertheless, the current implementation is usable and provides TLS 1.3 secured communication for the strongSwan project.

## 2. Introduction

This thesis is structured into six chapters: This introduction gives an overview of the topic in which the thesis is set. The Transport Layer Security (TLS) protocol is discussed broadly, the Open Source project strongSwan is introduced and how strongSwan uses the TLS protocol is elaborated. The chapter ends with a definition of the project scope for this thesis.

In the second chapter the background theory of the covered TLS 1.3 topics is explained. This part focuses mainly on RFC 8446 and illustrates and describes all the important handshakes TLS 1.3 supports and which are covered in this work. It also examines how TLS 1.3 negotiates the supported protocol versions and describes how the new HMAC-based key derivation function is designed.

The third chapter focuses on concrete concepts that are used either in the strongSwan code base or how the protocol acts in real world scenarios. It will give a brief architecture overview of the strongSwan TLS library `libtls` and discuss some general implementation design choices we made for this work. After this we discuss key aspects of this work that are absolutely mandatory to understand before the actual implementation of these features can take place. These are especially how PSK and PSK-based session resumption works in TLS 1.3, how the HKDF state machine is designed for this project and finally how and when session keys and IVs are derived and set for inbound and outbound traffic.

The implementation chapter gives an overview of the features that were implemented during this thesis. We emphasise some essential code fragments that are important to the goals of this work. We also point to the individual commits which implement each discussed feature.

The achieved implementation was extensively tested. Unit tests guaranteed basic functionalities were tested automatically. In addition, manual test cases were executed. The testing chapter focuses on the latter and describes how and what was tested.

Finally, we summarise the whole work done and the goals achieved during this bachelor thesis in the results chapter. Furthermore, we list some open tasks and future work that could be done in `libtls` and close with a brief overview of which direction TLS is going to evolve in.

We also add an extensive appendage that captures a list of abbreviations, the bibliography, instructions to compile and execute the code of this work as well as the references to the source code and the used OpenSSL commands for this work.

### 2.1. Overview

TLS is the protocol standard that secures connections between a client and a server. It is located between the transport layer (ISO OSI layer 4) and the application layer (ISO OSI layer

7) in the network stack (see figure 2.1).<sup>1</sup> The most popular TLS version 1.2 was defined in 2008 (RFC 5246), but only found broad adoption around 2014.<sup>2</sup> In that same year, work on TLS 1.3 started. After several revisions, it was written down in RFC 8446 (2018), with significant changes from its predecessor. It took thus roughly ten years to introduce a new version of the TLS protocol.

The strongSwan project spans a similar time frame and secures communications on networks as well. Its IPSec implementation authenticates and encrypts network traffic on the network layer (ISO OSI 3). Originally, strongSwan was a 2005 fork of the discontinued FreeS/WAN project by John Gilmore.<sup>3</sup> The project features its own TLS stack, yet with client and server support only up to TLS version 1.2. A client-side prototype stack with support for TLS 1.3 was implemented in a preliminary work<sup>4</sup> and improved in this work. The present work aims to complete the implementation of RFC 8446 as required by strongSwan.

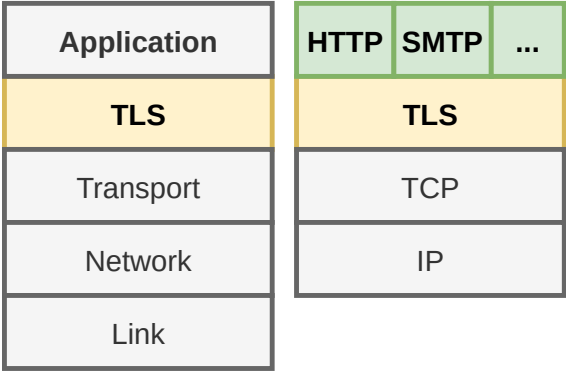


Figure 2.1.: The location of TLS 1.3 in the four layer TCP/IP reference model defined by the ARPA of the DoD.

## 2.2. Transport Layer Security Protocol (TLS)

The Transport Layer Security protocol (TLS) secures communication between client and server and is one of the fundamental elements in today's Internet. It encrypts and authenticates data from higher-level protocols such as the Hypertext Transfer Protocol (HTTP), and guarantees that the information transmitted from one endpoint to another remains confidential and unmodified.<sup>5</sup>

Communication between client and server is achieved by exchanging messages of the *Record Protocol*, which in itself encapsulates the *Handshake Protocol*, the *Application Data Protocol* to encapsulate actual payload data and the *Alert Protocol*. The three latter sub-protocols are layered on top of the record protocol as illustrated in figure 2.2.<sup>6</sup>

In a typical TLS 1.2 session, the client initiates a connection with a `ClientHello` message and sends its cryptographic capabilities to the server, notably the protocol version and cipher suites it supports. The server sends its own information back in several messages and signals the end of the negotiation with a `ServerHelloDone`. Then the client sends its cryptographic

<sup>1</sup>If not indicated otherwise, all illustrations are our own.

<sup>2</sup>17.

<sup>3</sup>20.

<sup>4</sup>8.

<sup>5</sup>Cf. [1, p. 236] and [6, p. 876].

<sup>6</sup>Some sources[19, p. 190][26] also describes *ChangeCipherSpec* as a sub-protocol of TLS. In RFC 8446 *ChangeCipherSpec* is described as message type and not as a protocol. It is although noted that the record protocol is typed and specifies six `ContentTypes`: `invalid`, `change_cipher_spec`, `alert`, `handshake`, `application_data` and `heartbeat`. [13, p. 122]

information in a `ClientKeyExchange` message. Note that the messages exchanged up to this point are still sent in plaintext as seen in figure 2.3.

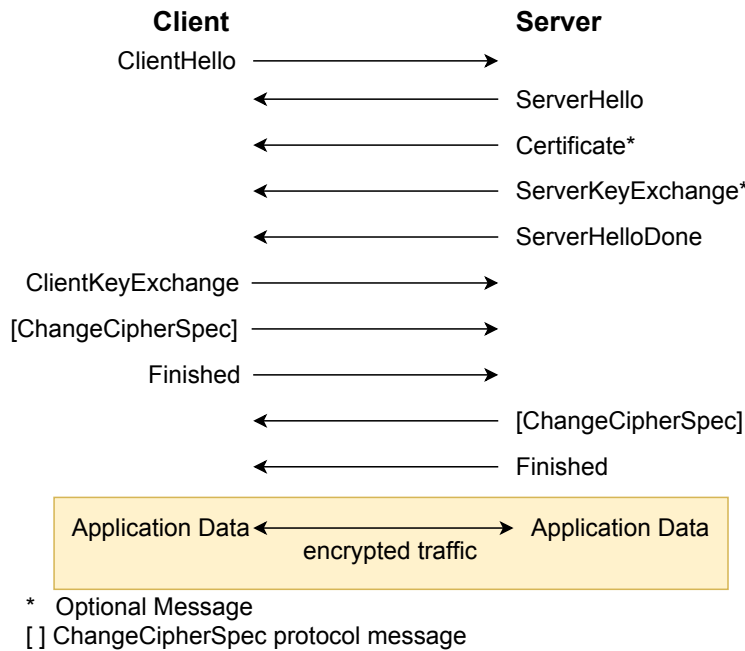


Figure adapted from from Ristić 2017:27

Figure 2.3.: Full handshake in TLS 1.2 with server authentication.

If the handshake was successful so far, enough cryptographic material is available to encrypt the following traffic. The client signals this state with a `ChangeCipherSpec` message. A `Finished` message concludes the handshake. After the server has responded with the same messages, secured application data can flow between the two endpoints, sent using the record protocol.<sup>7</sup>

TLS 1.2 was an important step towards securing communication over the Internet, yet it has also inherited several features and design choices from its predecessor versions.<sup>8</sup> The handshake flow above is only one out of many and the complexity of TLS 1.2 made its implementation prone to errors, bugs and security issues.<sup>9,10</sup> The current version of TLS 1.3 presents a major overhaul of the

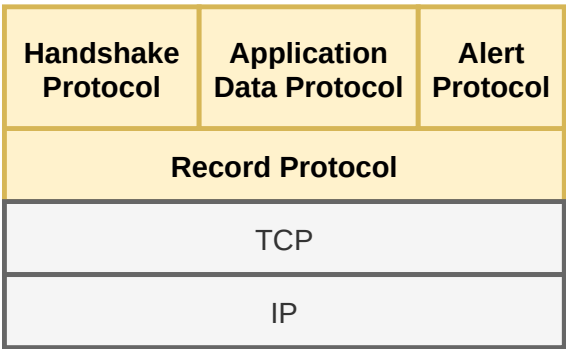


Figure 2.2.: Network protocol stack with the four TLS sub-protocols (yellow).

<sup>7</sup>We presume a basic knowledge of TLS here and note only the most important aspects here. A very good introduction to TLS 1.2 can be found in [15].

<sup>8</sup>1, p. 237.

<sup>9</sup>A cursory search lists 655 security vulnerabilities in Mitre’s CVE database, including Crime (2012), the notorious Heartbleed (2014), Poodle (2015) and other attacks. <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=TLS>, visited 2020-10-10.

<sup>10</sup>All major TLS development steps as well as the most important security flaws can be found in the “SSL/TLS and PKI History” <https://www.feistyduck.com/ssl-tls-and-pki-history/>, visited 2020-10-20.

previous TLS version 1.2. The two most notable changes relate to the handshake process, and the encryption and authentication schemes to be used.

The handshake, which establishes a secure connection between client and server, has been greatly simplified in TLS 1.3. This becomes most apparent in comparison with the state machines of the two TLS versions: The official specification does not list any state machine at all, with the various possible states in different handshakes flows.<sup>11</sup> On the other hand, TLS 1.3 devotes a whole appendix chapter that lists the various possible state machines.<sup>12</sup>

In comparison to figure 2.3, server and client exchange fewer messages:

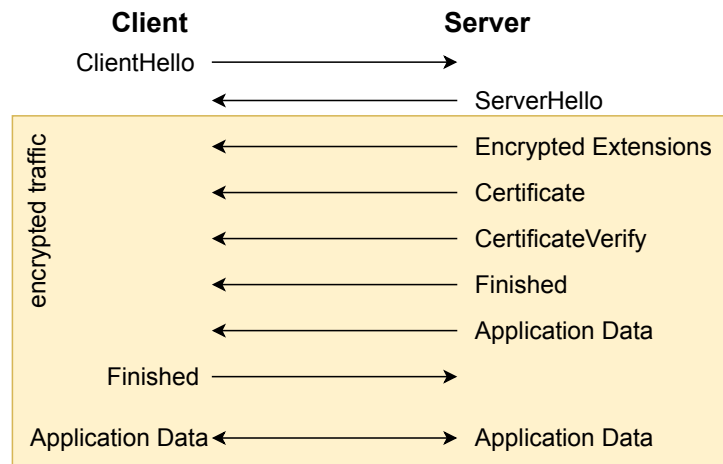


Figure 2.4.: Full handshake in TLS 1.3 with server authentication.

The client basically announces “Hello, I want to establish a connection” and provides the necessary key material, the server answers with its own key material and other information needed, and the handshake is over. Essentially, this is a major speed-up, since it eliminates a full round-trip between client and server. The `ChangeCipherSpec` message, which formerly announced the transition to encrypted application data, is now obsolete. It has only been preserved in TLS 1.3 to improve interoperability with non-standard behaviour of middleboxes such as firewalls and proxies and is not processed further.<sup>13</sup> In addition to this speed-up, TLS 1.3 even allows a zero-round-trip (0-RTT) connection establishment if the client has previously connected to the server.

The other important improvement in figure 2.4: Data is encrypted much earlier in the whole client-server message exchange. The encryption happens right after the `ServerHello`, since the key material is ready earlier than in TLS 1.2. All following information such as the server certificate are now encrypted, while being sent in plaintext in previous TLS versions.

What is not visible in figure 2.4 is the expanded use of extensions. They already existed in previous TLS versions, but with the latest one a lot of information has been moved to new or pre-existing extensions, notably also the supported TLS versions of client and server.

The major change in TLS 1.3 is only visible under the hood, however: Encryption algorithms, hashing and signature algorithms have been pruned and reorganised. This change is most

<sup>11</sup>2.

<sup>12</sup>13, p. 120.

<sup>13</sup>13, pp. 77, 140.

visible in the list of cipher suites of which a client can offer a selection to the server. The page-long cipher suite combinations<sup>14</sup> have been reduced to five recommended cipher suites.<sup>15</sup> Cipher suites in TLS 1.2 are of the following format, with an example just below:

TLS\_[key exchange]\_[authentication]\_with\_[cipher]\_[mac or prf]  
TLS\_ECDHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256

This is thus a cipher suite that uses elliptic curve ephemeral diffie-hellman (ECDHE) for the key exchange, RSA for authentication, 128 bit advanced encryption standard (AES) in galois/counter mode (GCM) mode as bulk cipher for encryption and decryption of the application data, and SHA-256 as hash function. TLS 1.3 mandates the exclusive use of *authenticated encryption with associated data* (AEAD)<sup>16</sup> algorithms. Currently only five of them exist thus we have a drastic reduction of cipher suites. For a more detailed description of AEAD see:<sup>17</sup>

TLS\_[AEAD]\_[hash]  
TLS\_AES\_128\_GCM\_SHA256

This is the 128 bit AES-GCM authenticated encryption algorithm together with a SHA256-based pseudo-random function (PRF).<sup>18</sup> The key exchange and signature algorithm field have been moved to their own respective extension. In addition, the algorithms for key exchange all support forward secrecy. If an attacker gains the private key of a server or client, they are not able to decrypt past captured TLS sessions.

Another notable change is the key derivation: TLS 1.3 uses a new HMAC-based Key Derivation Function (HKDF) that has been defined in RFC 5869<sup>19</sup> in 2010.

TLS 1.3 brings many new advantages and implies a quick adoption. Holz et al. note that a couple of providers and big Internet corporations were able to quickly implement TLS 1.3, e.g. Cloudflare, Google and Facebook.<sup>20</sup> Yet smaller providers have been rather slow in adopting the new TLS version as the authors note. This work aims to lift the adoption of TLS 1.3 further and let all strongSwan users use the new protocol version.

### 2.2.1. TLS 1.3

TLS 1.3 has many major and minor changes compared with older TLS versions. We list some of the more important changes regarding this thesis here. A more elaborate list is found in RFC 8446.<sup>21</sup>

---

<sup>14</sup>The IANA lists over more than 300 cipher suites: <https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml>, visited on 2020-10-20.

<sup>15</sup>13, p. 133.

<sup>16</sup>We briefly discuss AEAD in section 4.4.1

<sup>17</sup>1, p. 148.

<sup>18</sup>To be more precise: The hashing algorithm is used to build an HMAC (Keyed-Hash Message Authentication Code) within the HKDF (Hash-based Key Derivation Function). The concept behind this is explained in section 3.7.

<sup>19</sup>9.

<sup>20</sup>7.

<sup>21</sup>13, p. 8.

- Supported symmetric cipher suites were reduced to non-legacy ciphers which all support AEAD.
- Cipher suite concept has changed and does not include authentication and key exchange algorithms any more.
- All public key based key exchange now provide forward secrecy.
- All handshake messages after the `ServerHello` are now encrypted.
- The key derivation function has been redesigned.
- The handshake state machine has been greatly redesigned.
- TLS 1.2 version negotiation mechanism has been deprecated.
- Session resumption has been replaced by a single new PSK exchange.

## 2.3. strongSwan

The strongSwan project is an open-source IPsec implementation. Originally, it was based on FreeS/WAN, but was completely rewritten in an object-oriented coding style in the programming language C.<sup>22</sup> IPsec features so-called “Security Associations” (SA), with which two peers establish security attributes, for example encryption algorithm and keys. In strongSwan, IKE and IKEv2 (Internet Key Exchange) are the protocols to create these Security Associations. In strongSwan, the `charon` library implements the IKEv2 protocol and acts as a keying daemon.<sup>23</sup> Since strongSwan version 4.5.0, `charon` supports EAP-TLS<sup>24</sup> to mutually authenticate client and server with X.509 certificates.<sup>25</sup>

### 2.3.1. TLS in strongSwan

The Extensible Authentication Protocol (EAP)<sup>26</sup> is a framework that allows a broad set of methods to authenticate peers. In terms of TLS, the methods EAP-TLS, EAP-TTLS and EAP-PEAP are relevant:

**EAP Transport Layer Security (EAP-TLS)** as defined in RFC 5216 uses the handshake protocol of TLS to authenticate both peers with certificates, while the TLS encryption functionality itself is not used.<sup>27</sup> A Public Key Infrastructure (PKI) is required and must be able to provide a client certificate.<sup>28</sup>

---

<sup>22</sup>21.

<sup>23</sup>23.

<sup>24</sup>18.

<sup>25</sup>22.

<sup>26</sup>25.

<sup>27</sup>19, p. 165.

<sup>28</sup>6, p. 838, Note that EAP-TTLS and PEAP do not need client certificates for authentication.



**EAP Tunneled Transport Layer Security (EAP-TTLS)** as defined in RFC 5281 first establishes a TLS connection between client and server and then authenticates each other in a second step over this encrypted channel. A certificate is only mandatory for the server, but optional for the client. After the secured tunnel is established, other authentication mechanisms can be used such as EAP.<sup>29</sup>

**Protected EAP (PEAP)** as developed by Cisco Systems, Microsoft and RSA Security<sup>30</sup>, is similar to EAP-TTLS and only requires a server-side certificate and optionally client-side certificate to establish a secure TLS connection. Afterwards EAP messages are sent encrypted over the connection.<sup>31</sup>

strongSwan itself implements a range of EAP methods and currently supports all TLS methods that are offered by EAP and listed above. The TLS library of strongSwan is further used for the “Posture Transport Protocol over TLS” (PT-TLS).<sup>32</sup> These use cases are the main reason why a TLS stack is present in strongSwan.

## 2.4. Project Scope

This is a follow-up work of the term project “TLS 1.3 for strongSwan: A Client-Side Prototype”<sup>33</sup> which was accomplished in the previous term.

This bachelor thesis builds directly on the achievements reached within the previous work and has the goal to enhance the missing features to get a solid TLS 1.3 stack usable for strongSwan’s needs.

The goals for this thesis are:

**Server implementation** The current TLS implementation in strongSwan covered the client- and the server-side up to TLS 1.2. Since large parts of the client-side TLS 1.3 implementation were already achieved in the aforementioned term project it is mandatory to achieve the server-side TLS 1.3 implementation in this thesis. This is required to facilitate integration testing also for the TLS 1.3 protocol in the strongSwan project.

**Mutual authentication** The current TLS implementation supports mutual authentication up to TLS 1.2. It is a mandatory goal for this thesis to implement this also for TLS 1.3 because the use case of TLS in strongSwan is to authenticate clients with the EAP protocols that relay on TLS. This also covers additions and improvements in the client-site TLS 1.3 implementation.

**PSK-based resumption** The current TLS implementation makes use of the session resumption capabilities older TLS versions provide. Such a feature is especially useful in cases where several sequential connections are used between the same client-server pair. The speed-up of the connection establishment is notable since CPU intensive cryptographic key generation can be omitted. In TLS 1.3 these old session resumption features are

---

<sup>29</sup>19, p. 165.

<sup>30</sup>[https://en.wikipedia.org/wiki/Extensible\\_Authentication\\_Protocol#PEAP](https://en.wikipedia.org/wiki/Extensible_Authentication_Protocol#PEAP), visited 2020-10-20.

<sup>31</sup><https://security.stackexchange.com/questions/147344/eap-tls-vs-eap-ttls-vs-eap-peap/> 149643, visited on 2020-10-20.

<sup>32</sup>16.

<sup>33</sup>8.

completely redesigned. It is therefore an optional goal in this thesis to also implement PSK-based session resumption.

As mentioned above, the term project prior to this thesis covered the implementation of a client-side prototype of a TLS 1.3 stack<sup>34</sup>. Concerning the documentation, this introduction and the description of the HKDF in 3.7 and 4.4 were taken over and improved. During this bachelor thesis several important additions were implemented, various factors improved and fixed on all components of the predecessor work.

### **2.4.1. Out of Scope**

The TLS specification is comprehensive and covers a broad set of application scenarios. It is not the goal to implement the full RFC 8446 specification. In this thesis we will focus on the strongSwan relevant topics, as elaborated in 2.4. For example, the feature “Early Data” is not of interest for strongSwan since it is not the goal to facilitate performant data transmission over TLS but rather to cover authentication related aspects.

---

<sup>34</sup>d5e7aea9: libtls: Implement HKDF for TLS 1.3 and 8f097fbb: libtls: Implement TLS 1.3 handshake on client-side

## 3. Theory

This chapter lays the theoretical foundation for this thesis. It focuses heavily on RFC 8446.

First, the basic concept of TLS handshakes is captured and the notation for the illustrations is described. Then, the usual TLS handshake scenarios are briefly described. These cover the regular 1-RTT handshake, the client authenticating mutual handshake as well as the PSK-based session resumption handshake. Also a handshake scenario with incorrect DHE shares is described. Since the TLS version negotiation process has changed in TLS 1.3 we address this in a separate section. Lastly, we describe the theoretical background of the HMAC-based key derivation function which replaces the way keys are generated in TLS 1.3 compared to its predecessor versions.

### 3.1. TLS handshakes

In TLS the handshake protocol is the most important one. Its purpose is to establish a secure channel between two peers. Therefore we discuss some of the important properties of the TLS 1.3 handshake protocol in this chapter. Different types of handshake communication are illustrated by several figures.

First, a regular TLS 1.3 handshake is described. The second handshake type is a mutual authentication communication, where a client authenticates itself to the server. Next, we discuss the pre-shared key (PSK) based handshake between two peers that already had a connection to each other in the past. Finally, we describe how a handshake needs an additional round in case the two peers have no common key exchange algorithm in the first messages.

Each TLS handshake is started by a client which sends a `ClientHello` message to a server. The handshake communication establishment consists of the three phases *Key Exchange*, *Server Params* and *Authentication*.

**Key Exchange** In this phase shared keying material is established between both peers. All messages after this phase are encrypted with the handshake traffic secret.

**Server Params** Communicate whether a client needs to authenticate itself or establish other handshake parameters.

**Authentication** Authenticates the server and optionally the client and verifies the integrity of the handshake.

In the illustrations in this chapter, differently shaped boxes are used to emphasise the type and purpose of messages and extensions sent between both peers. These shapes are explained in figure 3.1.

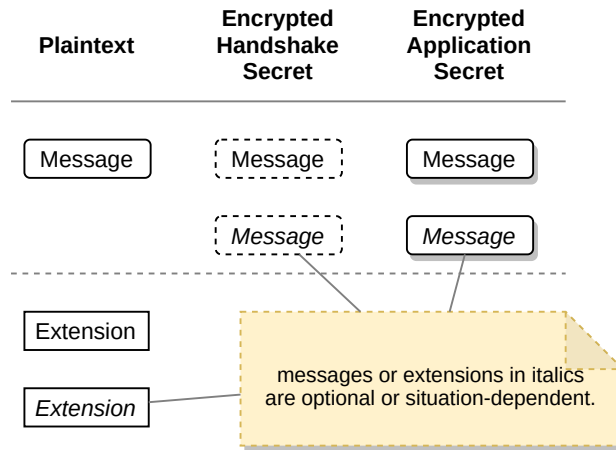


Figure 3.1.: Explanation of shape use for messages and extensions.

### 3.2. 1-RTT TLS 1.3 handshake

This generic type of a handshake covers most of the messages and extensions which can be used in a TLS handshake. We describe the mandatory messages and the message flow briefly here.

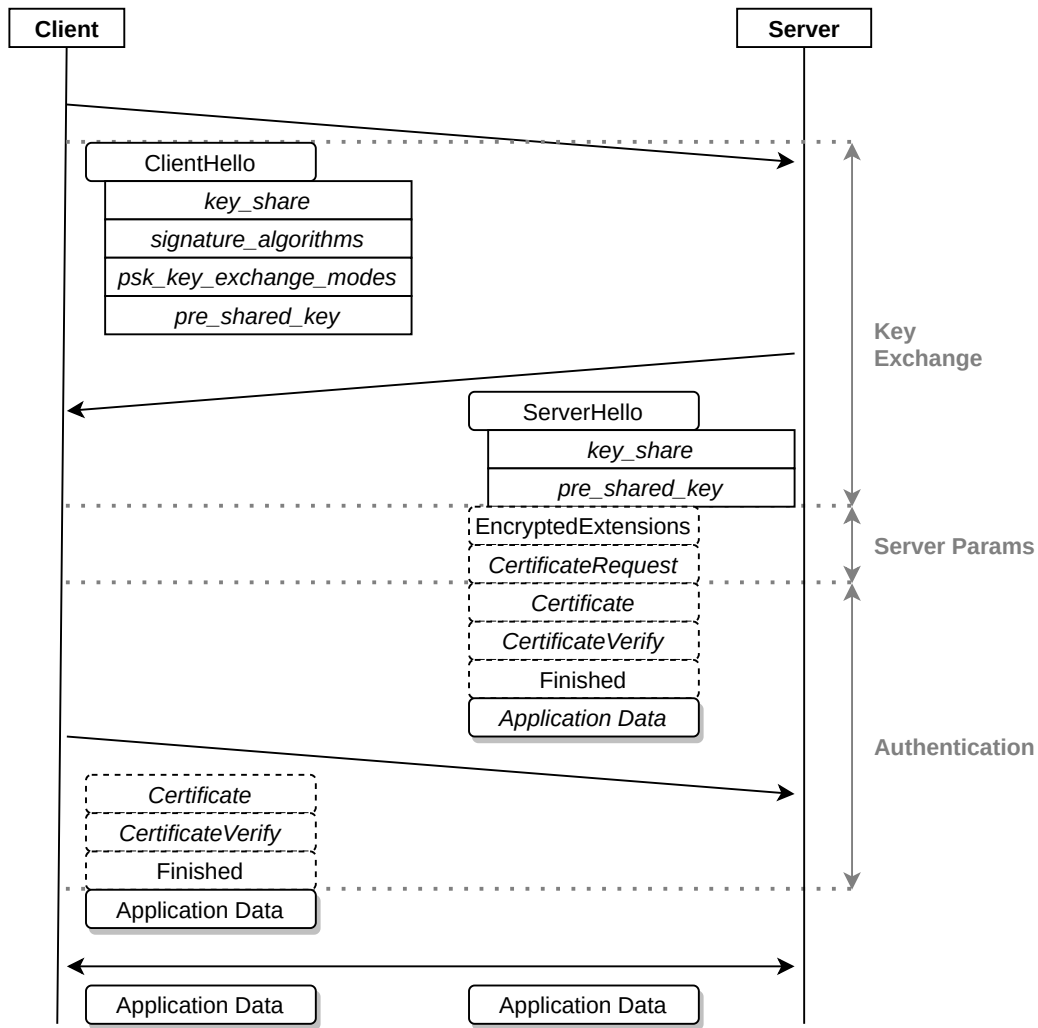


Figure 3.2.: Regular 1-RTT TLS 1.3 handshake.

The client initiates the handshake with a `ClientHello` message and several extensions. The server parses the appropriate extensions and message properties and responds with a `ServerHello` message with a subset of unencrypted extensions.

Now both parties share the same cryptographic key material and are able to derive the `handshake_traffic_secret`. Next, the server sends the `EncryptedExtensions` message followed by the `Finished` message. The latter builds a cryptographic signature over the whole handshake so far. Also this message indicates the end of the handshake from the server's point of view. The client parses the `ServerHello` message and switches after this to the `handshake_traffic_secret` to successfully decrypt `EncryptedExtensions` and `Finished` messages.

The client verifies the signature received in the `Finished` message. It also sends a cryptographic signature from its perspective of the whole handshake within a `Finished` message back to the server. Both peers derive the `application_traffic_secret` and switch the keys after each sent the `Finished` message. Application data can be sent by each peer as soon as the corresponding `Finished` has been sent and the `handshake_application_secret` keys are derived. Section 4.5 describes key derivation and switching in detail.

### 3.3. Mutual authentication TLS 1.3 handshake

In this handshake the server asks the client to authenticate itself with a X.509 certificate.

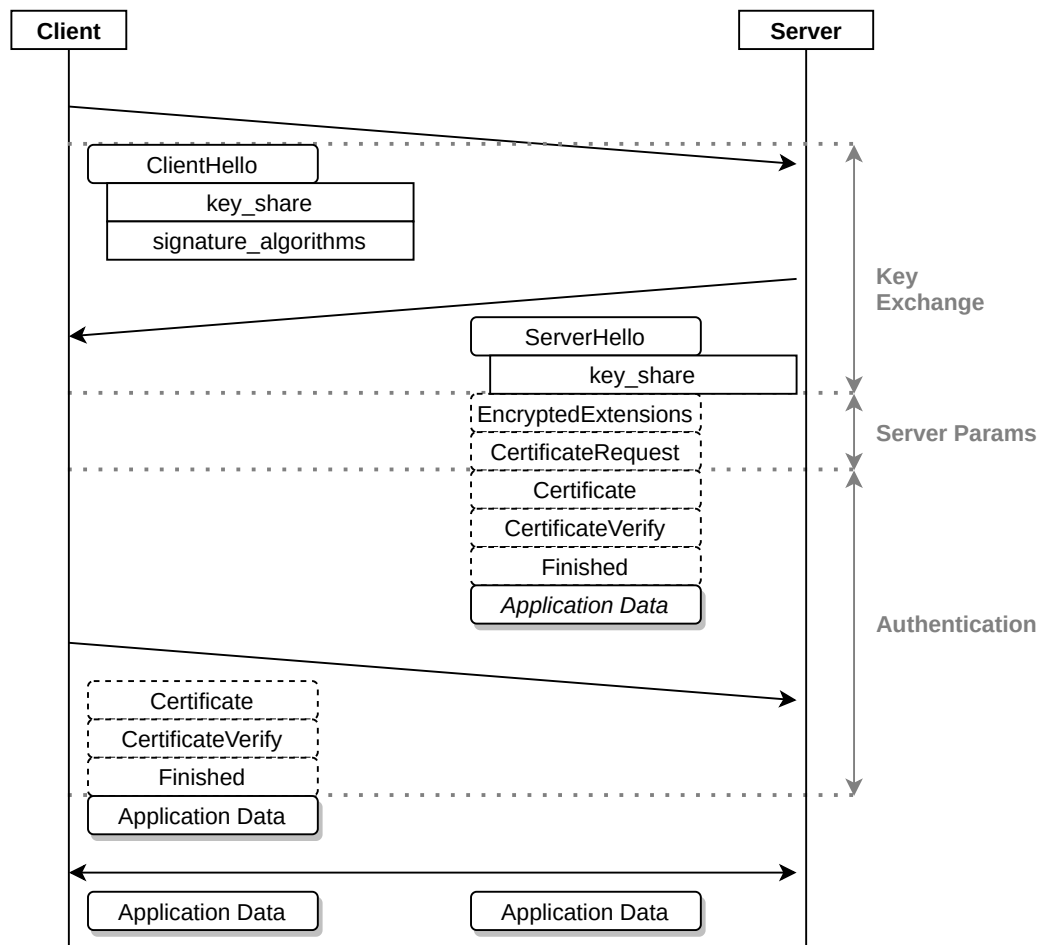


Figure 3.3.: Mutual authentication TLS 1.3 handshake.

Whereas in figure 3.2 the `CertificateRequest` message was optional it is mandatory now. The message has to be sent by the server directly after the `EncryptedExtensions` and just before the `Certificate` message. The client is now requested to authenticate itself. It does so by sending a `Certificate` and `CertificateVerify` just before the `Finished` message back to the server. The server verifies the signature from within the `CertificateVerify` message with the client certificate from the client's `Certificate` message.

### 3.4. PSK session resumption TLS 1.3 handshake

TLS 1.3 provides two separate ways to use PSKs: Either use an external generated PSK on both peers or establish a connection bases on the PSK derived in a previous TLS session. This chapter describes the latter which is also known as “session resumption” or “session resuming” with PSK.

The way how TLS sessions can be reused in a follow-up connection has changed significantly in TLS 1.3 compared to predecessor versions. In TLS 1.2 and below this was achieved by using the “session ID” and “session tickets”. Both of these handshake fields are obsolete in TLS 1.3.<sup>1</sup>

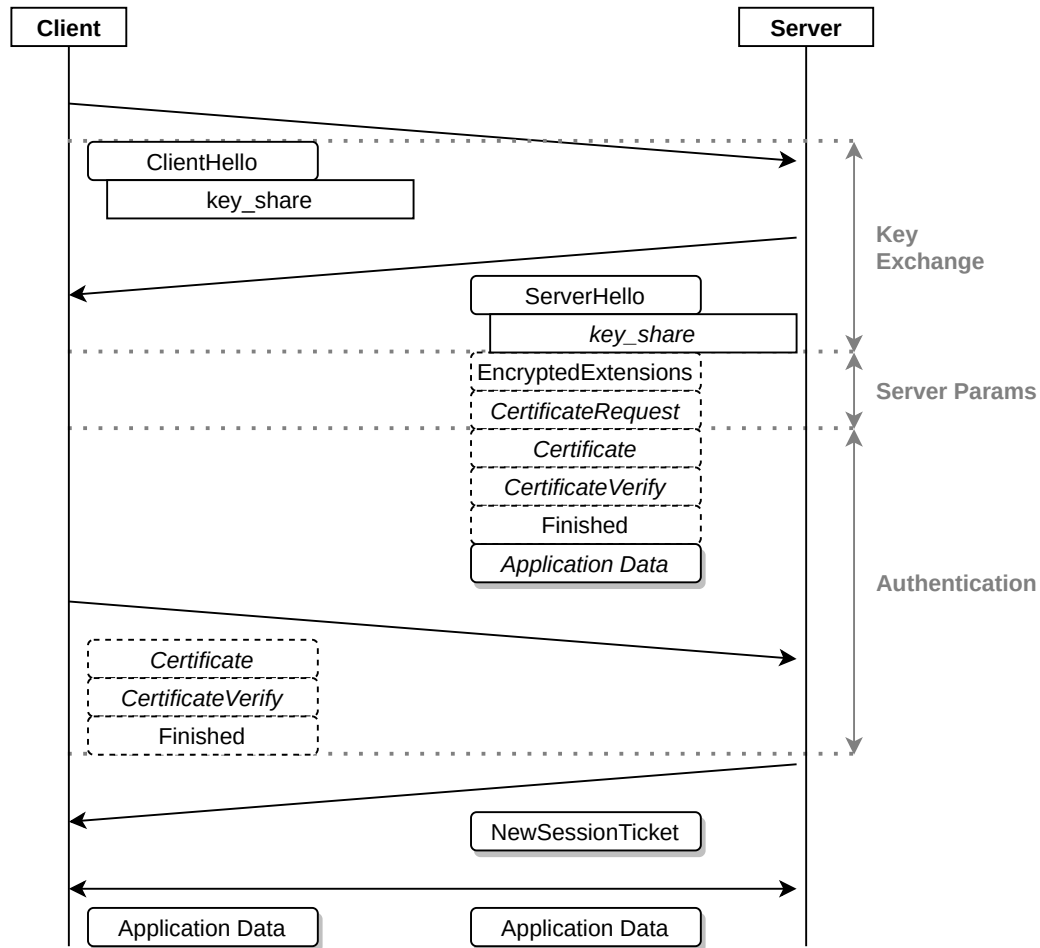


Figure 3.4.: Initial session to use PSK in the following TLS 1.3 connection.

After a TLS 1.3 connection is established the server may send one or multiple `NewSessionTicket` messages over the secured channel to the client as illustrated in figure 3.4. Its content allows both peers to speed-up the subsequent connection.

<sup>1</sup>13, p. 15.

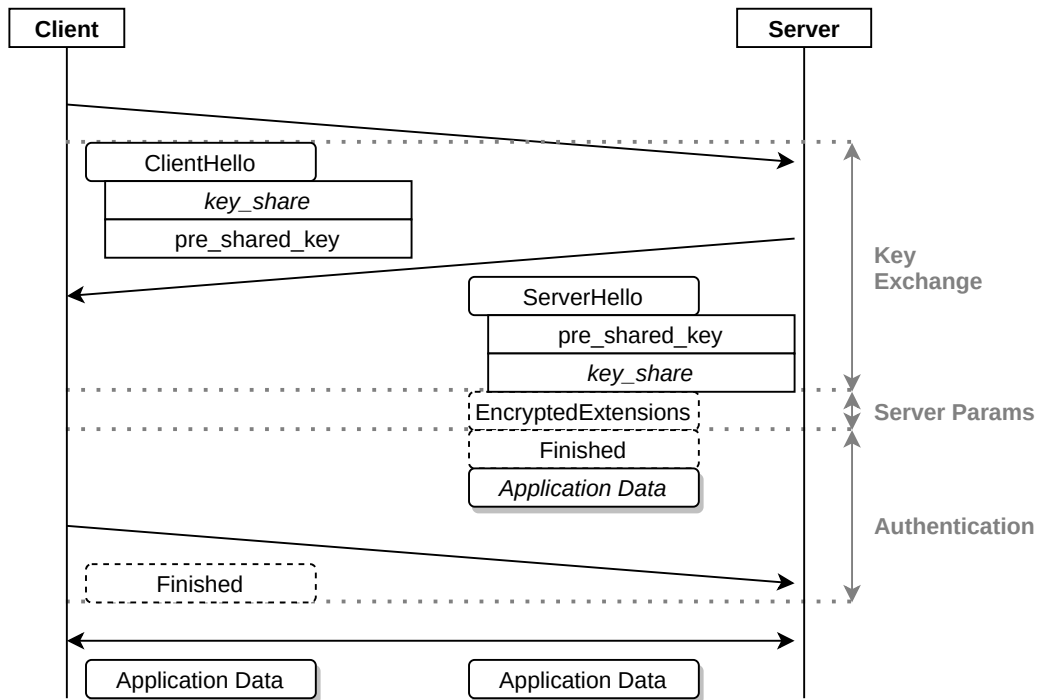


Figure 3.5.: PSK session resumption TLS 1.3 handshake.

If the client in the future opens a new connection to the server and supports PSK-based session resumption it sends the data received from the server in the first connection in the **pre\_shared\_key** extension within the **ClientHello** message as described in figure 3.5.<sup>2</sup> The client should also send a regular **key\_share** extension in case the server is not able to proceed with the PSK session resumption mode and a fall back to a regular full handshake is needed.

The server also sends back a **pre\_shared\_key** in its **ServerHello** message to negotiate the PSK key. It can also send a **key\_share** extension to provide *forward secrecy* for the current connection.<sup>3</sup> However, no authentication messages such as **Certificate** and **CertificateVerify** are used in the second handshake.

### 3.5. Incorrect DHE Share

If a client sends a key share generated with a DH group, e.g. *secp256r1* but the server does not support this DH group, it may send a **HelloRetryRequest** as response to the **ClientHello**. In this message the server asks for an alternative DH group, e.g. *x25519*, based on the values the client sent in the **supported\_groups** extension in the initial message.

The client follows the instruction and generates new asymmetric key material and sends the new public key again in a second **ClientHello** message. From this point onward the handshake is as described in 3.2.

<sup>2</sup>Noteworthy the **pre\_shared\_key** extension must be the last extension within the **ClientHello** message.

<sup>3</sup>To negotiate the wish for forward secrecy, the extension **psk\_key\_exchange\_modes** is used.



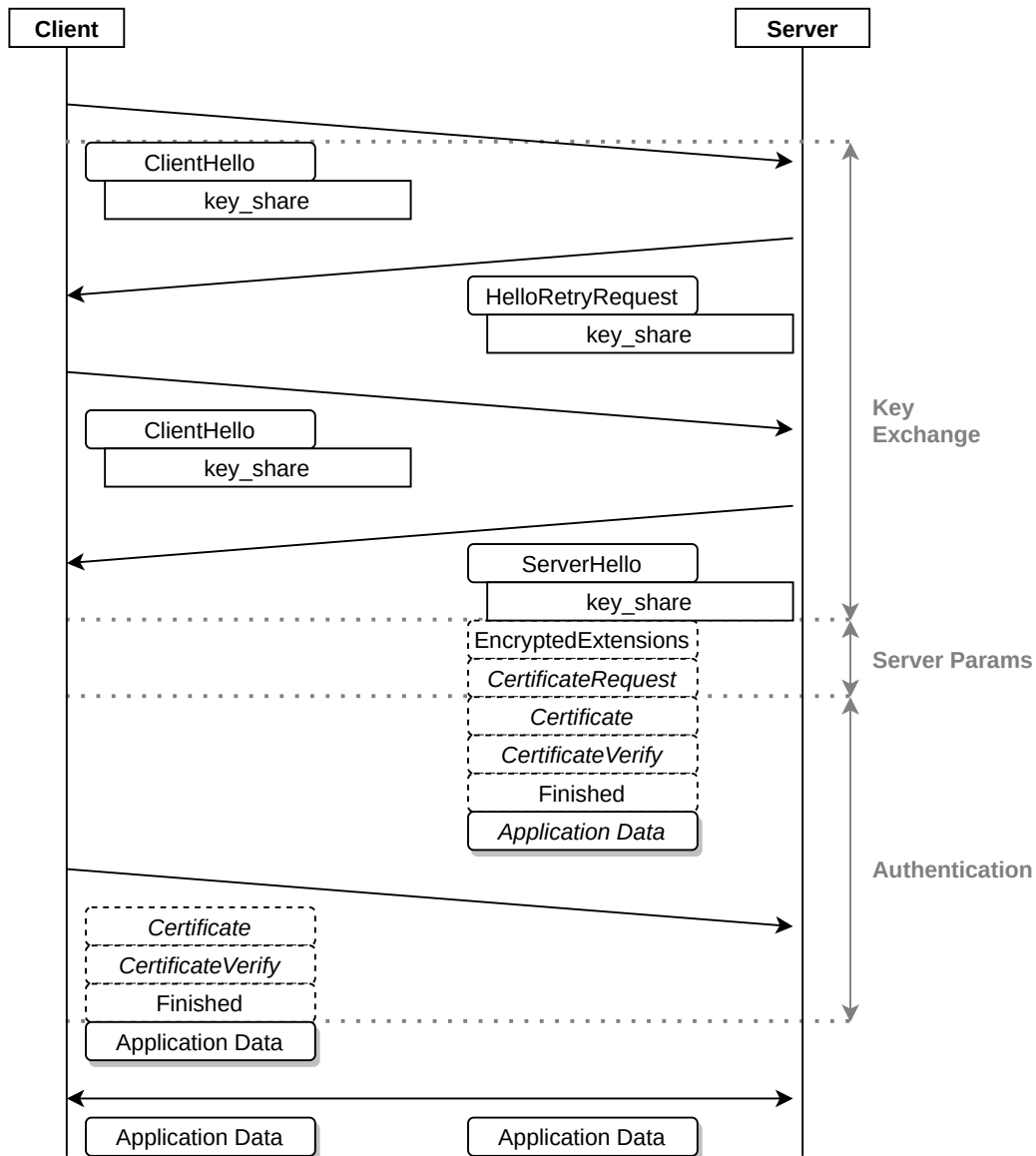


Figure 3.6.: Incorrect DHE Share.

### 3.6. TLS 1.3 version negotiation

In all TLS versions the *supported version negotiation* between client and server is exchanged in the first two messages. The client offers the versions it is able to speak and the server chooses its maximal capable version.

In contrast, in predecessor versions of TLS 1.3 the negotiation took place directly in the ClientHello respectively ServerHello handshake messages as “version” field in which the client sets the two byte TLS version value it maximally supports. The server answered with the same field and value if it agreed or suggested a smaller version value if it was not able to speak the same TLS version as the client.

In TLS 1.3 the “version” fields in the two messages have been deprecated and the version negotiation has been moved to the extensions in form of the `supported_version` extension.

This allows the client to send a list of TLS versions it speaks. The server chooses one of the offered values.

RFC 8446 also reflects this change back to TLS 1.2 implementation. If an implementation does support TLS 1.3 it should also be able to negotiate TLS 1.2 by using the `supported_version` extension.

### 3.7. HMAC-based Key Derivation Function

In TLS it is necessary to generate keys which encrypt the traffic exchanged by the peers. However, the algorithm differs between TLS versions. TLS 1.2 uses a pseudo-random function (PRF) to generate keys from the Master-Key.<sup>4</sup> TLS 1.3 on the other hand uses an HMAC-based Key Derivation Function (HKDF) to generate traffic keys.<sup>5</sup>

A HKDF has two fundamental functions: HKDF-Extract and HKDF-Expand.<sup>6</sup> However, TLS 1.3 introduces two new additional functions, HKDF-Expand-Label and Derive-Secret.<sup>7</sup> This results in these four functions:

- HKDF-Extract extracts a pseudo-random key (PRK) from the source key<sup>8</sup>. The extraction function is based on an HMAC, hence the name HKDF. This function takes two parameters: An input key material IKM and a salt.
- HKDF-Expand is a second step in which the generated pseudo-random key is fed to an HMAC. The HMAC in turn acts as a pseudo-random function to extract the required amount of bits. This function takes three parameters: A Secret, a HkdfLabel and the desired output key material (OKM) Length.
- HKDF-Expand-Label transforms Label and Context into a HkdfLabel structure<sup>9</sup> and calls HKDF-Expand to derive an OKM. This function takes the four arguments Secret, Label, Context and Length.
  - Context contains the handshake messages of the current state or an empty string as a hash.
  - Length specifies the desired output length in bytes.
- Derive-Secret hashes the raw handshake message bytes and calls HKDF-Expand-Label to derive an OKM. This function takes the three arguments Secret, Label and Messages.
  - Secret: The PRK secret from the HKDF-Extract function.
  - Label: An indicator to generate a specific secret. All relevant labels are defined in 3.7.1.

---

<sup>4</sup>14, p. 26.

<sup>5</sup>13, p. 95.

<sup>6</sup>9.

<sup>7</sup>The basic concepts of an HKDF are described in chapter 7.1, “Key Schedule”, and chapter 7.3, “Traffic Key Calculation” [13, 91ff].

<sup>8</sup>Dan Boneh has an excellent video explaining HKDF in detail. See [4].

<sup>9</sup>13, p. 91.

- Messages specifies the unencrypted handshake bytes, without record headers, of client and server up to the current state of the TLS handshake.

The hashing algorithm used for the HKDF is specified in the negotiated TLS cipher suite. Since the number of cipher suites in TLS 1.3 is reduced to five, only two possible hashing algorithms currently remain: SHA-256 or SHA-384.

Figure 3.7 illustrates how the HKDF is used in TLS, and is directly drawn from the illustration in RFC 8446.<sup>10</sup> It suggests that the HKDF in TLS 1.3 can also be interpreted as a state machine: Starting from a phase 0, every call to HKDF-Extract is a one-way transition into the next phase. This results in a state machine with four phases.<sup>11</sup>

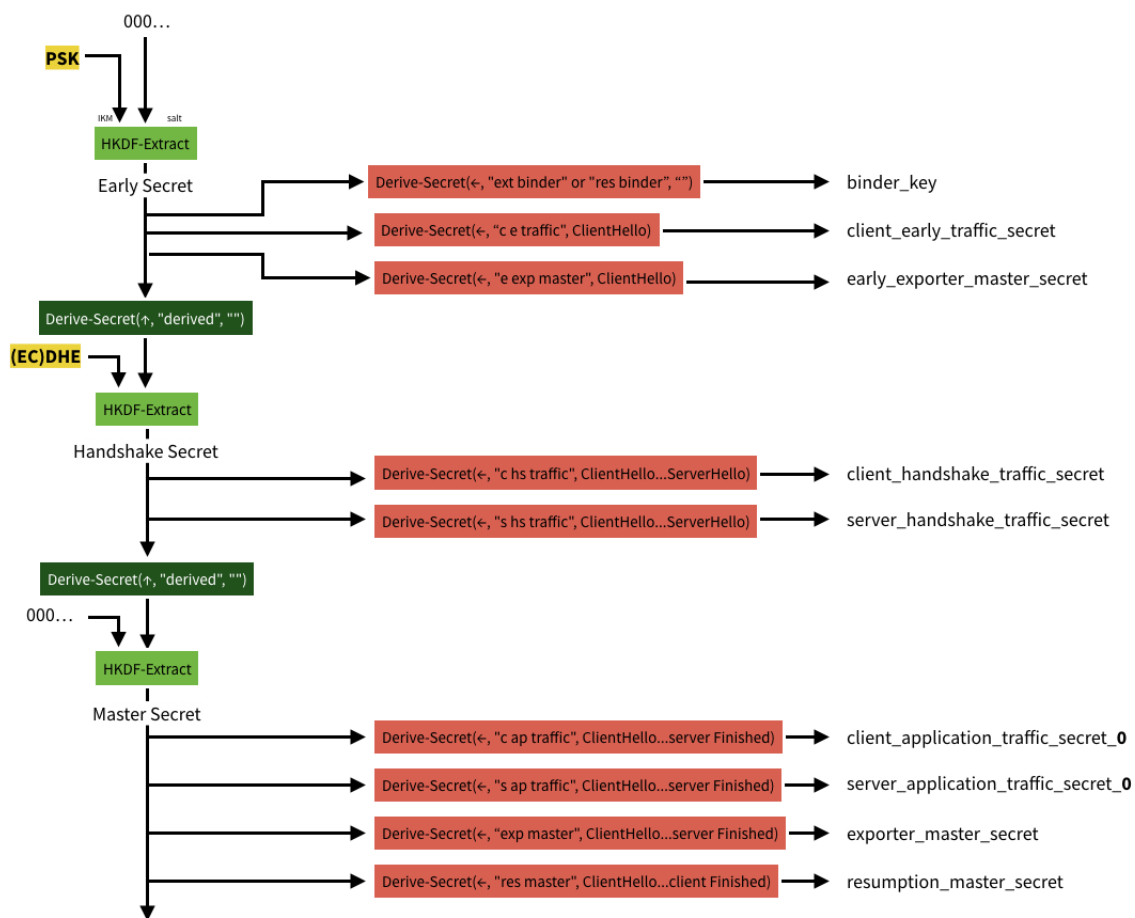


Figure 3.7.: Each HKDF-Extract signals a one-way state transition.

### 3.7.1. HKDF Labels

The above-mentioned labels play an important role in the HKDF. Unfortunately, RFC 8446 does not provide a single overview and description of all possible labels. It is therefore important to mention these other labels and where they can be used as well.

<sup>10</sup>13, p. 93.

<sup>11</sup>The illustration in figure 3.7 is a courtesy of David Wong, <https://www.davidwong.fr/tls13/#section-7.1>, visited on 2020-12-1.

As mentioned, eleven labels are defined in section 7.1 of the RFC and associated to different stages in the HKDF state machine.<sup>12</sup> The label “tls13 derived” is only used during state transitions. These labels are all passed as arguments to the Derive-Secret function:

The four phase 1 labels are all derived from the *Early Secret*:

- “tls13 ext binder” To get a *binder key* with an external provided PSK.<sup>13</sup>
- “tls13 res binder” To get a *binder key* for a resumption PSK.<sup>14</sup>
- “tls13 c e traffic” To get the *client early traffic secret* in PSK resumption handshake scenarios.<sup>15</sup>
- “tls13 e exp master” To get the *early exporter master secret* which provides cryptographic material for third-party protocols.<sup>16</sup>

The two phase 2 labels are all derived from the *Handshake Secret*:

- “tls13 c hs traffic” To get the *client handshake traffic secret* which is used to encrypt handshake data from the client to the server.
- “tls13 s hs traffic” To get the *server handshake traffic secret* which is used to encrypt handshake data from the server to the client.

The four phase 3 labels are all derived from the *Master Secret*:

- “tls13 c ap traffic” To get the *client application traffic secret* which is used to encrypt handshake data from the client to the server.
- “tls13 s ap traffic” To get the *server application traffic secret* which is used to encrypt handshake data from the server to the client.
- “tls13 exp master” To get the *exporter master secret* which provides cryptographic material for third-party protocols.<sup>17</sup>
- “tls13 res master” To get the *resumption master secret* which is used and stored locally for PSK-based session resumptions.<sup>18</sup>

There are a couple of labels that are used directly on HKDF-Expand-Label. These two provide actual keying material<sup>19</sup> and can be called on all states of the KDF state machine except phase 0:

- “tls13 key” To get the phase and secret corresponding encryption and decryption key.
- “tls13 iv” To get the phase and secret corresponding IV.

Generate the secret to authenticate the handshake Finished messages<sup>20</sup>:

---

<sup>12</sup>13, p. 93.

<sup>13</sup>13, p. 94.

<sup>14</sup>13, p. 94.

<sup>15</sup>13, p. 73.

<sup>16</sup>13, p. 97.

<sup>17</sup>13, p. 97.

<sup>18</sup>13, pp. 75, 92.

<sup>19</sup>Section 7.3 of [13, p. 95].

<sup>20</sup>Section 4.4.4 of [13, p. 71].

- “tls13 finished”

Key and initialisation vector (IV) are used in the AEAD ciphers to actually encrypt and decrypt traffic, be it for the handshake or application data. The label “tls13 finished” is only called once on each side, after a `Finished` message is received.

To generate actual PSK resumption secrets, the label “tls13 resumption” is defined and used directly on the `HKDF-Expand-Label` function with the *resumption master secret* from phase 2. Multiple resumption secrets can be generated by using a per ticket nonce.<sup>21</sup>

---

<sup>21</sup>13, p. 75.

## 4. Concept

This chapter covers strongSwan's TLS design concept as well as the more practical aspects of the theory discussed in the last chapter.

First, an overview of the architecture of the strongSwan TLS library `libtls` is given. Second, some of the important implementation design choices for this thesis are elaborated. Then, we focus on more complex key aspects of TLS 1.3. In the section PSK, the new session resumption approach of TLS 1.3 is explained. Section HKDF describes how the state machine to derive keys was designed and, lastly, the topic of key switching during the handshake is covered.

### 4.1. Architecture Overview of libtls

The TLS stack of the strongSwan `libtls` library has been written from scratch based on the corresponding TLS RFCs. TLS version 1.0, 1.1 and 1.2 and most of the associated cipher suites and cryptographic primitives are currently supported.

Figure 4.1 shows which components rely on this TLS stack:

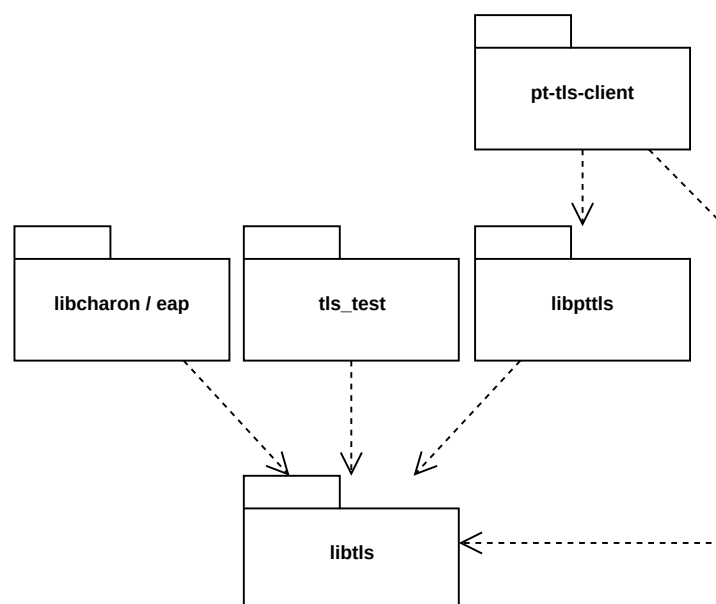


Figure 4.1.: Package diagram of the components `libcharon`, `libpttls` and `tls_test` which rely on `libtls`.

- The `eap` plug-in within the library `libcharon` uses TLS to implement mutual authentication over TLS (EAP-TLS, EAP-TTLS and PEAP).

- `tls_test` is a simple command-line executable that allows to set up a TLS connection as client or server. Once a TLS channel is set up, it allows each side to transfer data to each other similar to a telnet session.
- `libpttls` is the library that implements the Posture Transport Protocol over TLS (PT-TLS).
- `pt-tls-client` is a concrete client that uses `libpttls` and `libttls`.

The last three components use TLS in a “regular” way, which means a client builds an encrypted connection to a server and then communicates over this secured channel.

## 4.2. Implementation design choice

### 4.2.1. Protocol version separation

TLS 1.3 behaves differently over legacy TLS versions in many ways. For example, the state machine of both, client and server, was simplified by reducing the amount of possible states. This leads to the question of how this new TLS version shall be implemented in the existing code base.

In principle, two different approaches are possible: a) completely separate the new code from the existing code or b) integrate the new code in the existing code. Both have their advantages and disadvantages.

Approach a) leads to new implementations of existing functions specifically to TLS 1.3. As the programming language is C in an object-oriented style, the easiest way would be to generalise the common methods in a header file and create separate files for the existing legacy TLS implementation and the new TLS 1.3 implementation. Existing code would be only minimally touched but this in turn leads to much duplicate code.

Regarding approach b) the existing file structure would be mainly kept. Existing functions and methods which should behave differently in both legacy and future TLS versions would on trend grow in length since they need to distinguish between the sets of TLS versions. Also it has to be guaranteed that existing functionality is not affected by new code.

In this work we follow the design choice made in the preliminary work<sup>1</sup> and continue with approach *b* because existing code can be easier reused and code splitting over multiple files for the legacy TLS version and TLS 1.3 version can be omitted.

Listing 1 shows an example of how a version splitting can easily be done. A comparison like *greater than or equal* is absolutely possible because the TLS versions translate into a sequence of values in which newer protocol versions are larger than older ones, as seen in listing 2.<sup>2</sup> By using this *greater than or equal* another notable aspect is considered: Future versions of TLS 1.3 are more likely to behave similar to TLS 1.3 than legacy TLS versions and therefore already choose the branch that is more likely the one that is correct. This goes hand in hand with the fact that older TLS versions, such as TLS 1.0 and TLS 1.1 are

---

<sup>1</sup>g.

<sup>2</sup>strongSwan does not support SSL 2.0.

going to be deprecated soon.<sup>3</sup> This version branching allows to remove code related to those versions without affecting TLS 1.3 and newer versions code.

---

```
1 if (this->tls->get_version_max(this->tls) >= TLS_1_3)
2 {
3     /* new TLS 1.3 branch */
4 }
5 else
6 {
7     /* existing legacy TLS branch */
8 }
```

---

Listing 1: Example TLS version branching

---

```
1 /**
2  * TLS/SSL version numbers
3  */
4 enum tls_version_t {
5     SSL_2_0 = 0x0200, /* 512d */
6     SSL_3_0 = 0x0300, /* 768d */
7     TLS_1_0 = 0x0301, /* 769d */
8     TLS_1_1 = 0x0302, /* 770d */
9     TLS_1_2 = 0x0303, /* 771d */
10    TLS_1_3 = 0x0304, /* 772d */
11 };
```

---

Listing 2: TLS version numbers

#### 4.2.2. State machine separation

Towards the end of the preliminary work<sup>4</sup> of this thesis, it became more and more clear that the state machine, which initially was shared between all TLS versions, was getting too complicated. So it was separated into a state machine for legacy TLS versions and one for TLS 1.3. This made the program flow much more clear and easier to debug.

One of the key points of choosing this approach was the asymmetric character of the protocol versions: In legacy TLS versions, for example, the client is the first party who sends the Finished message whereas in TLS 1.3 the server sends this message first as illustrated in figures 2.3 and 2.4. Nevertheless, the new TLS 1.3 state machine shares the same state definition in the code for the common stats of both machines but adds its new state definitions as listed in listing 3.

---

<sup>3</sup>12.

<sup>4</sup>8.



---

```
1  typedef enum {
2      STATE_INIT,
3      STATE_HELLO_SENT,
4      STATE_HELLO_RECEIVED,
5      STATE_HELLO_DONE,
6      STATE_CERT_SENT,
7      STATE_CERT_RECEIVED,
8      STATE_KEY_EXCHANGE_RECEIVED,
9      STATE_CERTREQ_RECEIVED,
10     STATE_KEY_EXCHANGE_SENT,
11     STATE_VERIFY_SENT,
12     STATE_CIPHERSPEC_CHANGED_OUT,
13     STATE_FINISHED_SENT,
14     STATE_CIPHERSPEC_CHANGED_IN,
15     STATE_FINISHED_RECEIVED,
16     /* new states in TLS 1.3 */
17     STATE_HELLORETRYREQ_RECEIVED,
18     STATE_ENCRYPTED_EXTENSIONS_RECEIVED,
19     STATE_CERT_VERIFY_RECEIVED,
20     STATE_FINISHED_SENT_KEY_SWITCHED,
21     STATE_KEY_UPDATE_REQUESTED,
22     STATE_KEY_UPDATE_SENT,
23     STATE_CERT_VERIFY_SENT,
24 } peer_state_t;
```

---

Listing 3: TLS peer state machine states

## 4.3. PSK

This section provides, first, an overview of the message flow to describe the general concept of PSK-based resumption. Second, the generation of the necessary secrets is described in more detail.

### 4.3.1. Handshake Overview

A concrete PSK handshake consists of the following messages and contents. The shown example is achieved with an OpenSSL client and server as described in E.7.

```

▼ TLSv1.3 Record Layer: Handshake Protocol: New Session Ticket
  Opaque Type: Application Data (23)
  Version: TLS 1.2 (0x0303)
  Length: 250
  [Content Type: Handshake (22)]
▼ Handshake Protocol: New Session Ticket
  Handshake Type: New Session Ticket (4)
  Length: 229
▼ TLS Session Ticket
  Session Ticket Lifetime Hint: 7200 seconds (2 hours)
  Session Ticket Age Add: 2506124691
  Session Ticket Nonce Length: 8
  Session Ticket Nonce: 0000000000000000
  Session Ticket Length: 208
  Session Ticket: cbf4fd54ddd51e750138a54a1f1c35dd1a64327db6b7bfc0...
  Extensions Length: 0

```

Figure 4.2.: First NewSessionTicket message from the server.

```

▼ TLSv1.3 Record Layer: Handshake Protocol: New Session Ticket
  Opaque Type: Application Data (23)
  Version: TLS 1.2 (0x0303)
  Length: 234
  [Content Type: Handshake (22)]
▼ Handshake Protocol: New Session Ticket
  Handshake Type: New Session Ticket (4)
  Length: 213
▼ TLS Session Ticket
  Session Ticket Lifetime Hint: 7200 seconds (2 hours)
  Session Ticket Age Add: 523831374
  Session Ticket Nonce Length: 8
  Session Ticket Nonce: 0000000000000001
  Session Ticket Length: 192
  Session Ticket: cbf4fd54ddd51e750138a54a1f1c35dd95c56c0e892b847f...
  Extensions Length: 0

```

Figure 4.3.: Second NewSessionTicket message from the server.

After the handshake is completed, the OpenSSL server transmits two “New Session Ticket” messages shown in figure 4.2 and 4.3.

```

▼ Extension: pre_shared_key (len=321)
  Type: pre_shared_key (41)
  Length: 321
▼ Pre-Shared Key extension
  Identities Length: 219
▼ PSK Identity (length: 192)
  Identity Length: 192
  Identity: cbf4fd54ddd51e750138a54a1f1c35dd95c56c0e892b847f...
  Obfuscated Ticket Age: 523868374
▼ PSK Identity (length: 15)
  Identity Length: 15
  Identity: 436c69656e745f6964656e74697479
  Obfuscated Ticket Age: 0
  PSK Binders length: 98
  PSK Binders

```

Figure 4.4.: ClientHello message with pre\_shared\_key extension in the second connection.

In the second connection from the same client to the same server, the OpenSSL client sends the `pre_shared_key` extension with two *PSK Identities* as shown in figure 4.4. The first contains data from the first NewSessionTicket shown in figure 4.3.

```

    ▾ Extension: pre_shared_key (len=2)
      Type: pre_shared_key (41)
      Length: 2
    ▾ Pre-Shared Key extension
      Selected Identity: 0

```

Figure 4.5.: ServerHello message with pre\_shared\_key extension in the second connection.

The server responds with the chosen identity as shown in figure 4.5. It chose the first identity of the two identities provided by the client.

```

    ▾ TLSv1.3 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 128
    ▾ Handshake Protocol: Server Hello
      Handshake Type: Server Hello (2)
      Length: 124
      Version: TLS 1.2 (0x0303)
      Random: dde5af7bf44c06e600434b0b50d7c0332c4df85e585ebdda...
      Session ID Length: 32
      Session ID: 979c7334191a75e8d4391cd6949078a22cf5baa246efe51a...
      Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
      Compression Method: null (0)
      Extensions Length: 52
      ▸ Extension: supported_versions (len=2)
      ▸ Extension: key_share (len=36)
      ▾ Extension: pre_shared_key (len=2)
        Type: pre_shared_key (41)
        Length: 2
        ▾ Pre-Shared Key extension
          Selected Identity: 0
      ▸ TLSv1.3 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
      ▸ TLSv1.3 Record Layer: Handshake Protocol: Encrypted Extensions
      ▸ TLSv1.3 Record Layer: Handshake Protocol: Finished

```

Figure 4.6.: The whole ServerHello message in PSK mode in the second connection.

Figure 4.6 shows that the pre\_shared\_key extension is the last message within the ServerHello message.

Protocol	Length	Info
TLSv1.3	349	Client Hello
TLSv1.3	1400	Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate, Certificate Verify, Finished
TLSv1.3	146	Change Cipher Spec, Finished
TLSv1.3	321	New Session Ticket
TLSv1.3	305	New Session Ticket
TLSv1.3	94	Application Data
TLSv1.3	94	Application Data
TLSv1.3	674	Client Hello
TLSv1.3	307	Server Hello, Change Cipher Spec, Encrypted Extensions, Finished
TLSv1.3	146	Change Cipher Spec, Finished
TLSv1.3	305	New Session Ticket
TLSv1.3	94	Application Data
TLSv1.3	94	Application Data

Figure 4.7.: Both handshakes with all messages in PSK mode.

Figure 4.7 shows that the PSK handshake transmits less messages than the regular full handshake. In the first handshake, it needed five messages of a total of 2'191 TLS protocol bytes before the first data could be transmitted from the client to the server. The second handshake took four messages of a total of 1'168 TLS protocol bytes before the first data could be transmitted from the client to the server. In the above example handshake, this is

a reduction of 20% in the amount of messages and a reduction of 46.69% regarding TLS protocol payload.

RFC 8446 describes in section 2.2<sup>5</sup> the handshake in a “Resumption and Pre-Shared Key (PSK)” scenario and in section 2.3<sup>6</sup> the handshake in a “0-RTT Data” scenario. For both handshakes, the extension `key_share` is described as optional or situation-dependent.

RFC 8446 suggests to add the `key_share` in a PSK scenario to provide a fallback method to establish a secure connection, in case the server is unable to proceed in the PSK handshake path.<sup>7</sup> Another reason to add this extension is to provide forward secrecy in subsequent connections.<sup>8</sup>

In the two RFC 8446 sections, the PSK feature is described as a method to authenticate the peers to each other<sup>9</sup> and to provide a solution for 0-RTT (also called *early data*) support.<sup>10</sup> The CPU intensive `key_share` extension is suggested in both cases as a fallback solution or to provide forward secrecy.

This leads to the conclusion that PSK-based handshakes mainly reduce the amount of data transmitted by the peers by omitting the need for `CertificateRequest` and `CertificateVerify` messages as described above.

#### 4.3.2. Secret Generation

The simplified handshake is illustrated in figure 4.8 with two TLS sessions. The handshake was completed, therefore both peers are in the third HKDF state (the master secret was successfully generated).

---

<sup>5</sup>13, p. 15.

<sup>6</sup>13, p. 17.

<sup>7</sup>**rfc8846**.

<sup>8</sup>13, p. 19.

<sup>9</sup>13, pp. 16, 17.

<sup>10</sup>13, p. 17.

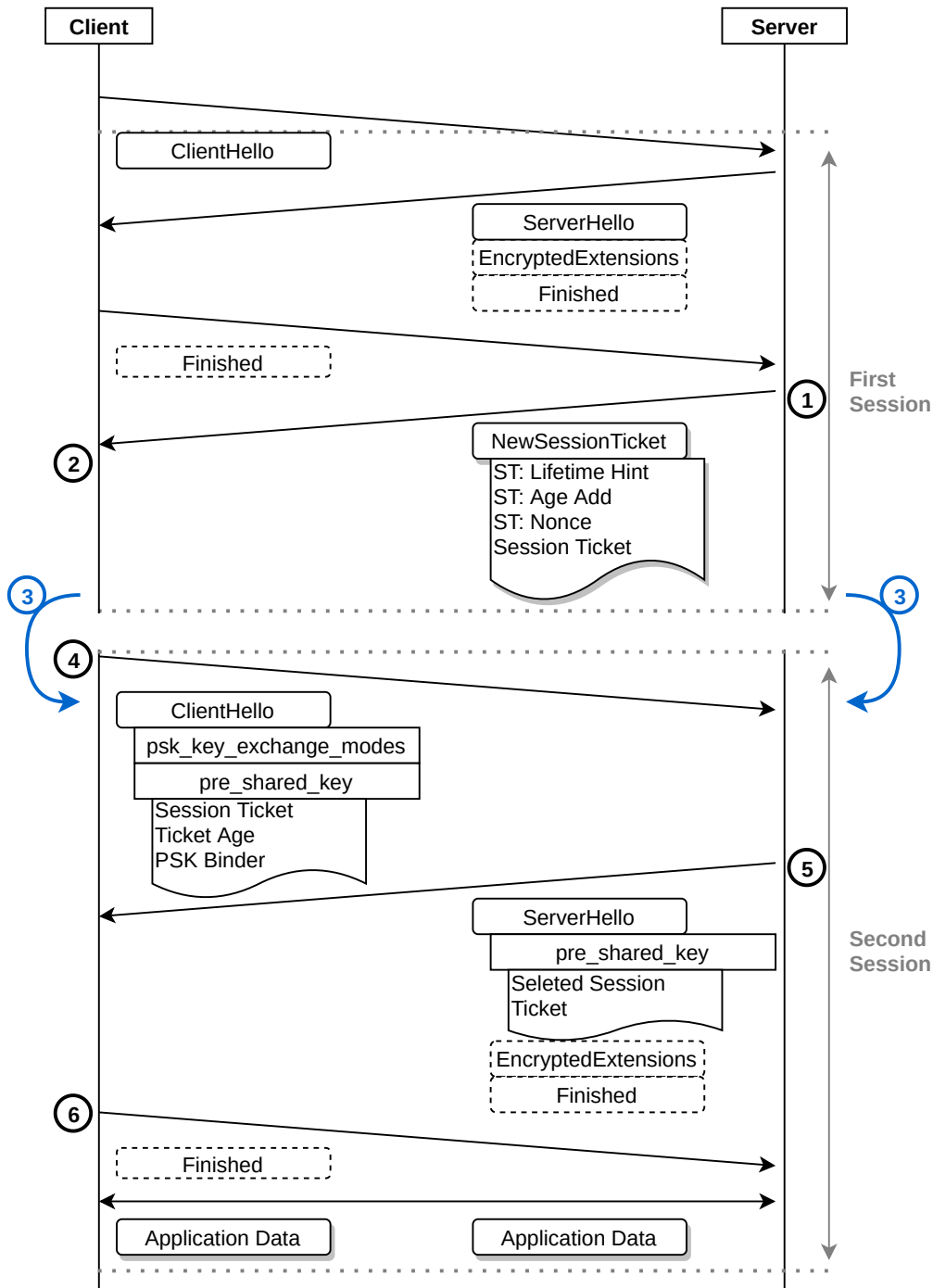


Figure 4.8.: PSK key generation.

1. The server generates the *resumption master secret* by using the label “tls13 res master” on its HKDF first and then generates a *session ticket*<sup>11</sup> for the NewSessionTicket message. The session ticket is an opaque label.<sup>12</sup> To associate a PSK with a session ticket, the HKDF is used with the label “tls13 resumption” together with a nonce. Also, two 32 bit integers need to be generated to indicate the lifetime of the ticket. Transmitted to the client within the NewSessionTicket message are *ticket lifetime*,

<sup>11</sup>In the illustration “ST” stands for “session ticket”.

<sup>12</sup>13, p. 75.

*ticket age add*, *ticket nonce* and *session ticket*. The PSK is not transmitted.

2. The client received the `NewSessionTicket` message and generates the PSK as described above. Because multiple tickets can be transmitted from the server to the client, each ticket has its own nonce. The client needs this nonce only to generate the same PSK.
3. This step takes place on either side of the connection. Both peers need to memorise the *session ticket*, the *ticket age add* and the generated PSK to reuse this information in a subsequent session. In other words, this is the only information that survives a session and can be used to establish a further connection.
4. The client sends two extensions in its `ClientHello` message: `psk_key_exchange_modes` and `pre_shared_key`. In the latter extension, the client sends the session ticket called *PSK identity* together with the *ticket age add* and the *PSK binder*. To get the *PSK binder*, the following steps are required: The client first initialises its HKDF with the PSK from the previous session and generates the *binder.key* by using the label “`tls13 res binder`”. Subsequently, it uses a hash of the `ClientHello` message it is about to send, until and with the PSK identities, to generate the *PSK binder*.<sup>13</sup>
5. A client could send multiple PSK identities and PSK binders, so the server finally chooses the identity to use and confirms this to the client.
6. The rest of the handshake is as already described in chapter 3.

## 4.4. HKDF

strongSwan did not bring a ready-to-use HKDF implementation the way TLS 1.3 requires. Such an implementation was one of the key achievements of the preliminary work<sup>14</sup> of this thesis. During this bachelor thesis the HKDF implementation was improved, enhanced and tested with additional unit tests. This section describes the design choices of the HKDF implementation used in this thesis.

The HKDF is used in different stages of the handshake process and functions like a state machine itself. Therefore, the design uses four distinct phases. They are fully encapsulated within the HKDF module (`tls_hkdf.h` and `tls_hkdf.c`). It is also important to understand that states move from lower phase numbers to higher ones as the arrows illustrate in figure 4.9.

The HKDF design chosen in this work has these four phases:

- Phase 0: the initial state
- Phase 1: generate pre-shared keys
- Phase 2: generate handshake keys
- Phase 3: generate application data keys

---

<sup>13</sup>At this stage not all length fields of the `ClientHello` message are set.

<sup>14</sup>8.

These phases have an internal flow, the logic is illustrated in figure 4.9. This design corresponds to the HKDF design as of RFC 8446 which was illustrated and discussed in figure 3.7.

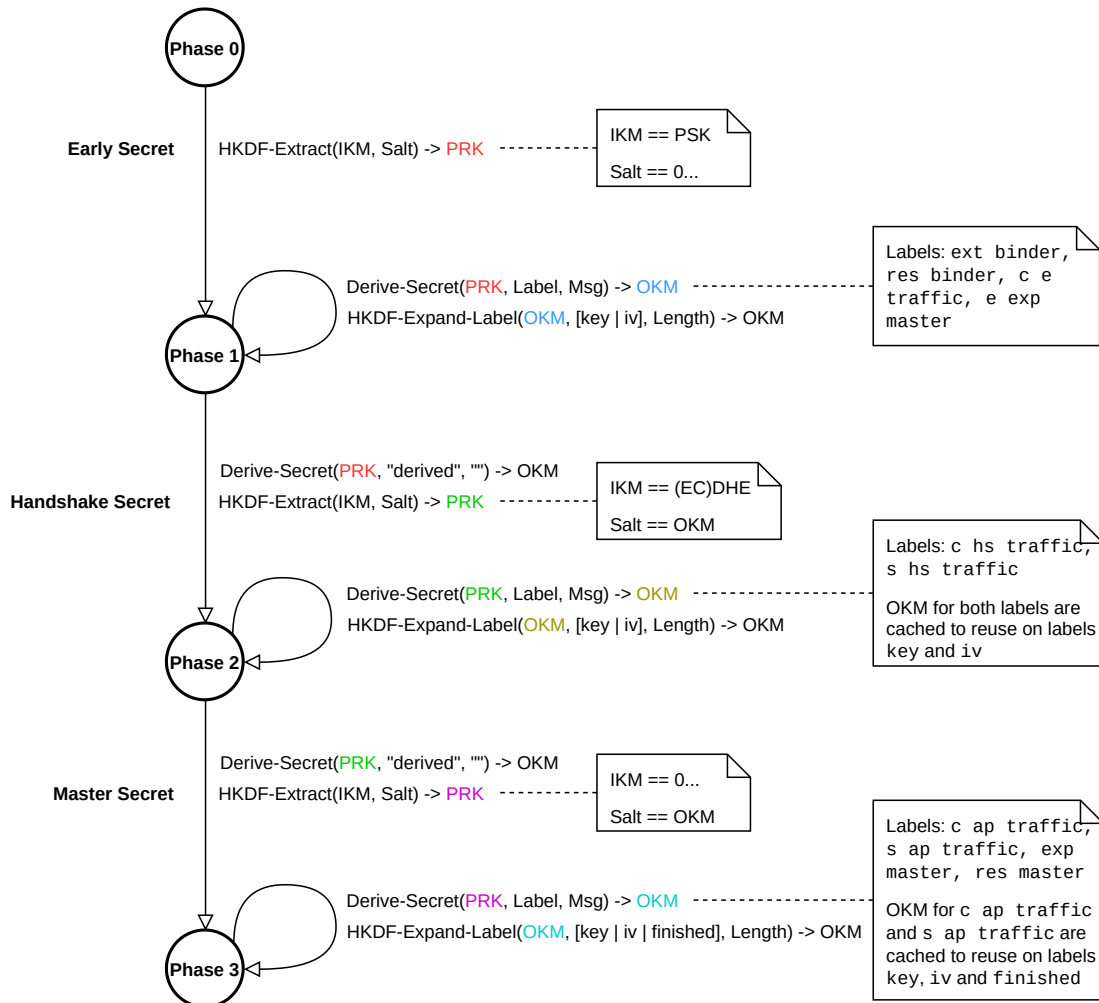


Figure 4.9.: The state machine for our HKDF implementation with the four phases.

These two aspects – repeated use and internal states – encourage a design which encapsulates the whole implementation in an HKDF class. The design of the public interface of the HKDF class is illustrated in figure 4.10:

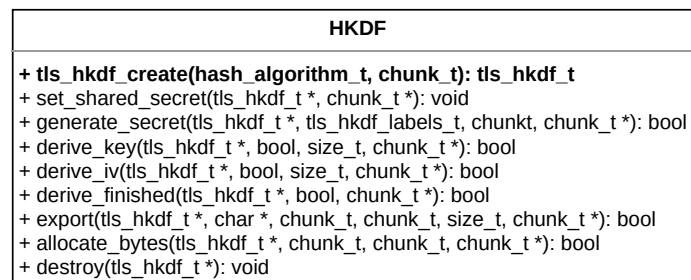


Figure 4.10.: UML class diagram of the current HKDF implementation.

Its functionality is verified by a broad set of unit tests (`test_hkdf.c`). The unit tests also give a good idea on how the class is used in the code. The constructor `tls_hkdf_create()` as well as all methods are documented using Doxygen, in accordance with the rest of the documentation.

The main idea of this design is to encapsulate all the HKDF functionality and internal state logic and provide an easier-to-use interface. A user may set a PSK secret when initialising a concrete HKDF object or leave it blank when not using secrets from the first phase. By providing one of the labels discussed in 3.7.1 and defined in `enum tls_hkdf_labels_t`, a user sets the HKDF state machine into the appropriate state using the function `generate_secrets`.

As one may notice from figure 4.9, a state transition always consists of the two steps Derive-Secret with the derived label and HKDF-Extract using the OKM from the former function. In the current implementation this fact is considered and these steps are encapsulated into the function `generate_secret`. Therefore, the label `derived` is not exposed to the caller and is solely used as internal label.

The states, as described earlier, provide different encryption keys and IVs derived with the corresponding public methods `derive_key` and `derive_iv`. To verify handshake authentication, the public method `derive_finished` is used.

While `derive_key` and `derive_iv` can be called in all three non-zero states, `derive_finished` is only called in phase 3 of the internal HKDF state machine.

#### 4.4.1. HKDF and AEAD

The idea behind AEAD is to encrypt the payload of a message and additionally authenticate the plaintext packet headers used to route the packet.<sup>15</sup> This allows the receiver to discover if the whole packet has been secretly modified in transit. All five cipher suites in TLS 1.3 are AEAD ciphers.<sup>16</sup> The key and initialisation vector (IV) are provided by the HKDF.

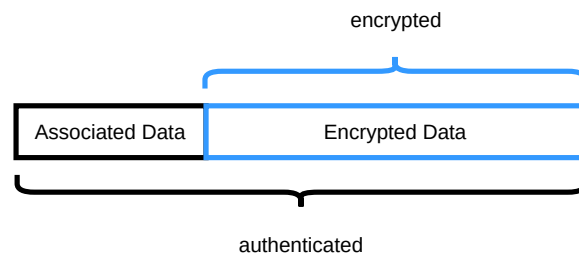


Figure 4.11.: Basic concept of authenticated encryption with associated data (AEAD).

strongSwan has already built-in support for AEAD, since TLS 1.2 standardised it.<sup>17</sup> However, TLS 1.3 uses AEAD somewhat differently, and one of the maintainers adapted the code to allow the current AEAD implementation to work with TLS 1.3. A new class was added, `tls_aead_seq.c`, and other changes happened in:

<sup>15</sup>See also illustration 4.11. It is heavily inspired by Dan Boneh's video explaining authenticated encryption.[3]

<sup>16</sup>This can also be recognised by the block cipher mode CCM and GCM respectively ChaCha20-Poly1305 for the stream cipher.

<sup>17</sup>14, p. 24.



- The public interface `tls_aead.h`
- All AEAD implementations of the interface:
  - `tls_aead.c`
  - `tls_aead.c`
  - `tls_aead_expl.c`
  - `tls_aead_impl.c`
  - `tls_aead_null.c`
- `tls_protection.c` with implements `tls_aead.h`

Because TLS 1.3 encrypts the actual content type within the `TLSInnerPlaintext` structure and always sets the `TLSCiphertext` structure content type to `Application Data`, the interface had to be modified.<sup>18</sup> This led to changes in all current existing AEAD implementations. Because `tls_protection.c` also uses the concrete AEAD object, it also required minimal changes there.

The new `tls_aead_seq.c` class is to be used with TLS 1.3 and its constructor has been added to the public interface in `tls_aead.h`. The class mainly differs in the way how key material is handled. According to RFC 8446, the nonce and initialisation vector (IV) is calculated differently than in the prior TLS version.<sup>19</sup>

## 4.5. Key Derivation and Switching

In section 4.4 the idea behind the HKDF design, functionality and implementation is described. For a concrete TLS 1.3 implementation, it is important to derive key material based on appropriate handshake data and at the correct state of the state machine. In figure 4.9 the key derivation and switching in a basic handshake is illustrated.

---

<sup>18</sup>13, p. 81.

<sup>19</sup>13, p. 83.

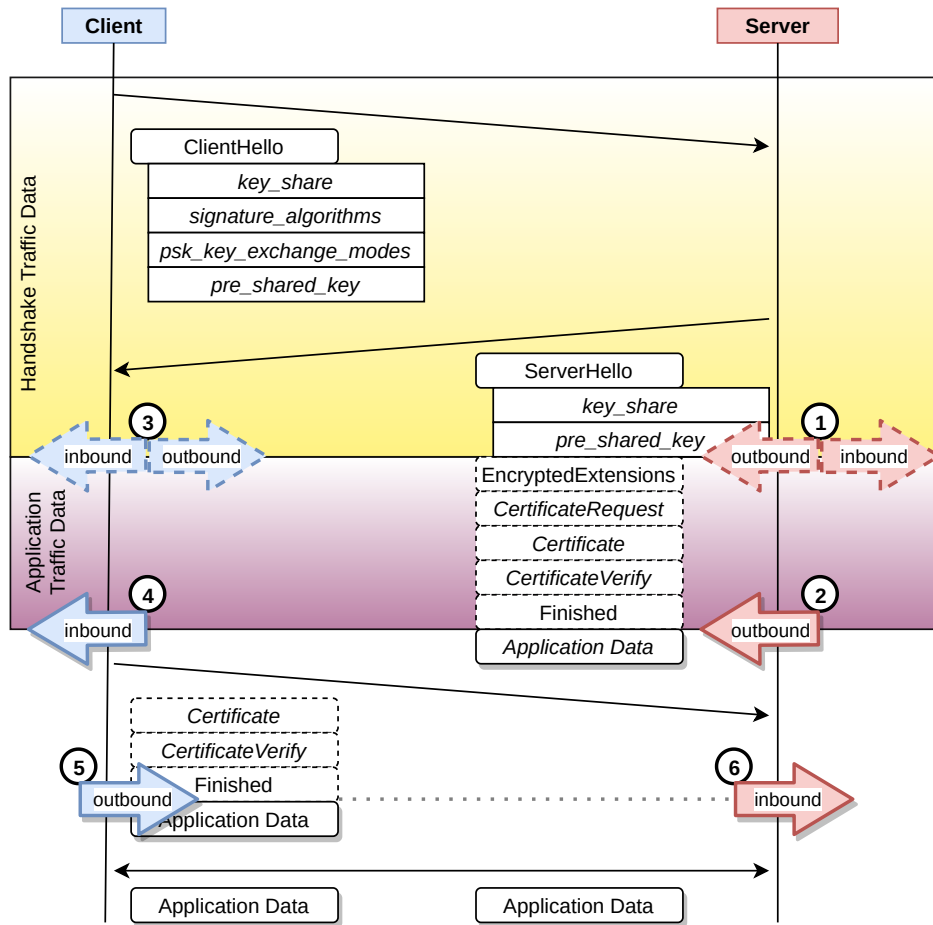


Figure 4.12.: Key material derivation and switching in TLS 1.3.

To derive the same keys, both client and server need the same handshake input data. The yellowish area indicates the handshake data to generate the `hs_traffic_secret`. This is the `ClientHello` message and the `ServerHello` message.<sup>20</sup> The purplish area highlights the handshake data to generate the `application_traffic_secret`. This area covers all messages up and including to the server-side `Finished` message. Both peers use this to derive their symmetric key material as follows:

1. The server generates its asymmetric key material and derives the common secret, the (EC)DHE secret, from it. The public key is sent to the client in the `ServerHello` message within the `key_share` extension. From this common secret a peer is able to generate the *handshake traffic secret* at this stage of the handshake. From this secret it now can derive and set keys and IVs for both channels “outbound” and “inbound”.
2. The server sends all following handshake messages, in this example `EncryptedExtensions` to `Finished`, encrypted with the key and IV derived from the first stage *handshake traffic secret*. After the `Finished` is sent the server may send already encrypted payload data which requires the *application traffic secret* from the second encryption stage to derive keys and IVs from. This is the reason why the server implementation at this

<sup>20</sup>A `HelloRetryRequest` message would also be within these two messages and therefore lies in this yellowish area.

stage has to switch to the second stage of encryption, the *application traffic secret*, and derive the key and IV for this stage and use it for “outbound” data.

3. The client parses the plain text `ServerHello` message and now possesses the public key of the server and therefore is able to get the common secret between the two peers in this session. From this (EC)DHE secret a peer is able to generate the *handshake traffic secret* and move to the first encryption stage respectively phase 1 in the HKDF state machine. From this *handshake traffic secret* it is now able to derive the keys and IVs to set both, the “inbound” and “outbound” channels. It is now able to encrypt the other messages up to the `Finished` message from the server.
4. As described in 2, it is possible<sup>21</sup> that the server already answers with encrypted payload at this stage, the client has to switch to the second encryption stage respectively phase 2 in the HKDF state machine by generating the *application traffic secret* and at this point needs to derive a key and IV for the “inbound” channel.
5. The client uses the *handshake traffic secret* of the first encryption stage to send its handshake messages, in this example `Certificate` to `Finished`, and switches to the second encryption phase on the “outbound” channel as soon as these messages are sent. For this switch it only needs to derive a key and an IV from the already generated *application traffic secret*.
6. The server derives and switches to the new key and IV on the “inbound” channel as soon as it received the client `Finished` message.

After these six steps, the handshake is completed and a secured session is established between both peers.

---

<sup>21</sup>in an early data scenario

## 5. Implementation

This chapter focuses on the concrete implementation achieved in this bachelor thesis. It discusses single code snippets of special interest. All topics reference to the involved commits which are publicly available on GitHub, in the repository described in D.

First, the server-side TLS 1.3 stack is described since this was the first goal achieved in this thesis. The second section focuses on the mutual authentication handshake which is the second goal achieved in this thesis.

### 5.1. Server-side implementation

Most changes and additions took place in `tls_server.c`, where the missing messages for building and processing were added. Most notable is, of course, the newly added state machine for processing, as listed in listing 4, and building, as in listing 5.<sup>1</sup>

---

<sup>1</sup>5edbe102: tls-server: TLS 1.3 support for TLS server implementation

---

```

1  if (this->tls->get_version_max(this->tls) < TLS_1_3)
2  {
3      /* removed */
4  }
5  else
6  {
7      switch (this->state)
8      {
9          case STATE_INIT:
10             if (type == TLS_CLIENT_HELLO)
11             {
12                 return process_client_hello(this, reader);
13             }
14             expected = TLS_CLIENT_HELLO;
15             break;
16         case STATE_CIPHERSPEC_CHANGED_IN:
17         case STATE_FINISHED_SENT:
18             if (type == TLS_FINISHED)
19             {
20                 return process_finished(this, reader);
21             }
22             return NEED_MORE;
23         case STATE_FINISHED_RECEIVED:
24             return INVALID_STATE;
25         default:
26             DBG1(DBG_TLS, "TLS %N not expected in current state",
27                 tls_handshake_type_names, type);
28             this->alert->add(this->alert, TLS_FATAL, TLS_UNEXPECTED_MESSAGE);
29             return NEED_MORE;
30     }
31 }

```

---

Listing 4: TLS 1.3 server-side process state machine.

Another important change was made in `tls_crypto.c`. `strongSwan` is able to verify signatures in both encodings, RSA and RSA-PSS, but only allows to sign in RSA-PSS. Therefore we had, at this stage of the project, to exclude all three RSA-PSS schemes, as shown on line 10 to 19 in listing 6.

---

```

1  if (this->tls->get_version_max(this->tls) < TLS_1_3)
2  {
3      /* removed */
4  }
5  else
6  {
7      switch (this->state)
8      {
9          case STATE_HELLO_RECEIVED:
10             return send_server_hello(this, type, writer);
11          case STATE_HELLO_SENT:
12          case STATE_CIPHERSPEC_CHANGED_OUT:
13             return send_encrypted_extensions(this, type, writer);
14          case STATE_ENCRYPTED_EXTENSIONS_SENT:
15             return send_certificate(this, type, writer);
16          case STATE_CERT_SENT:
17             return send_certificate_verify(this, type, writer);
18          case STATE_CERT_VERIFY_SENT:
19             return send_finished(this, type, writer);
20          case STATE_FINISHED_SENT:
21             return INVALID_STATE;
22          default:
23             return INVALID_STATE;
24      }
25  }

```

---

Listing 5: TLS 1.3 server-side build state machine.

---

```

1  /**
2   * Get the signature parameters from a TLS signature scheme
3   */
4  static signature_params_t *params_for_scheme(tls_signature_scheme_t sig)
5  {
6      int i;
7
8      for (i = 0; i < countof(schemes); i++)
9      {
10         /* strongSwan supports only RSA_PSS_RSAE schemes for signing but can
11          * verify public keys in rsaEncryption as well as rsassaPss encoding.
12          * Current implementation does not distinguish between signing and
13          * verifying. */
14         if (sig == TLS_SIG_RSA_PSS_PSS_SHA256 ||
15             sig == TLS_SIG_RSA_PSS_PSS_SHA384 ||
16             sig == TLS_SIG_RSA_PSS_PSS_SHA512)
17         {
18             continue;
19         }
20         if (schemes[i].sig == sig)
21         {
22             return &schemes[i].params;
23         }
24     }
25     return NULL;
26 }

```

---

Listing 6: Remove RSA-PSS signature schemes.

We later<sup>2</sup> enhanced the function signature to distinguish if a signing or verifying operation is required and allow RSA-PSS schemes for signing again, as shown in line 5 and 13 in listing 7.

---

```
1  /**
2   * Get the signature parameters from a TLS signature scheme
3   */
4  static signature_params_t *params_for_scheme(tls_signature_scheme_t sig,
5                                              bool sign)
6  {
7      int i;
8
9      for (i = 0; i < countof(schemes); i++)
10     {
11         /* strongSwan supports only RSA_PSS_RSAE schemes for signing but can
12          * verify public keys in rsaEncryption as well as rsassaPss encoding. */
13         if (sign && (sig == TLS_SIG_RSA_PSS_PSS_SHA256 ||
14                     sig == TLS_SIG_RSA_PSS_PSS_SHA384 ||
15                     sig == TLS_SIG_RSA_PSS_PSS_SHA512))
16         {
17             continue;
18         }
19         if (schemes[i].sig == sig)
20         {
21             return &schemes[i].params;
22         }
23     }
24     return NULL;
25 }
```

---

Listing 7: Allow RSA-PSS signature schemes for verifying.

### 5.1.1. KeyUpdate

KeyUpdate requests and responses are an important part in TLS connections to allow rekeying. Because the strongSwan TLS library is interoperable with any TLS library, this feature was also implemented<sup>3</sup> together with the TLS 1.3 server-side implementation.

The two new states `STATE_KEY_UPDATE_REQUESTED` and `STATE_KEY_UPDATE_SENT` were introduced in `tls_server.c` alongside with functions to generate and process these messages. In the process state machine, handling for messages of type `TLS_KEY_UPDATE` is inserted, as seen in listing 8. In the build state machine, the two new states are also covered, as of listing 9

---

<sup>2</sup>d9058b2a: tls-crypto: Distinguish between signing and verifying signature schemes

<sup>3</sup>2478aa03: tls-server: Support KeyUpdate requests and answers

---

```

1 case STATE_FINISHED_RECEIVED:
2     if (type == TLS_KEY_UPDATE)
3     {
4         return process_key_update(this, reader);
5     }

```

---

Listing 8: Server-side process state machine with KeyUpdate support.

---

```

1 case STATE_KEY_UPDATE_REQUESTED:
2     return send_key_update(this, type, writer);
3 case STATE_KEY_UPDATE_SENT:
4     if (!this->crypto->update_app_keys(this->crypto, FALSE))
5     {
6         this->alert->add(this->alert, TLS_FATAL, TLS_INTERNAL_ERROR);
7         return NEED_MORE;
8     }
9     this->crypto->change_cipher(this->crypto, FALSE);
10    this->state = STATE_FINISHED_RECEIVED;

```

---

Listing 9: Server-side build state machine with KeyUpdate support.

### 5.1.2. HelloRetryRequest (HRR)

HelloRetryRequest messages are an important part in TLS connections to negotiate DH groups. Because the strongSwan TLS library is interoperable with any TLS library, this feature was also implemented<sup>4</sup> together with the TLS 1.3 server-side implementation.

Multiple additions and changes had to be implemented in `tls_server.c` to achieve HRR support. We discuss some of the important aspects in short here.

From the server's point of view, it is important to know whether the client is currently on its first or second ClientHello message. Therefore, we added a new function to easily answer this question based on memorised session stats in form of the chunk `requested_curve` and the current state of the state machine as seen in listing 10.

---

```

1 /**
2  * Check if client is currently retrying to connect to the server.
3  */
4 static bool retrying(private_tls_server_t *this)
5 {
6     return this->state == STATE_INIT && this->requested_curve;
7 }

```

---

Listing 10: Server-side HRR retrying function.

The object field `requested_curve` is set in the static `process_client_hello()` function.

---

<sup>4</sup>a503a278: tls-server: Handle HelloRetryRequest (HRR)



## 5.2. Mutual authentication

We split the concrete mutual authentication support into two commits, first the server-side: New state `STATE_FINISHED_SENT_KEY_SWITCHED` was added that addresses the situation when the server `Finished` message is sent *and* the application traffic keys are in-place. The process state machine is listed in listing 11 and the build state machine in listing 12.<sup>5</sup>

---

<sup>5</sup>bcad44bc: tls-server: Mutual authentication support for TLS 1.3

---

```

1  if (this->tls->get_version_max(this->tls) < TLS_1_3)
2  {
3      /* removed */
4  }
5  else
6  {
7      switch (this->state)
8      {
9          case STATE_INIT:
10             if (type == TLS_CLIENT_HELLO)
11             {
12                 return process_client_hello(this, reader);
13             }
14             expected = TLS_CLIENT_HELLO;
15             break;
16         case STATE_CIPHERSPEC_CHANGED_IN:
17         case STATE_FINISHED_SENT:
18         case STATE_FINISHED_SENT_KEY_SWITCHED:
19             if (type == TLS_CERTIFICATE)
20             {
21                 return process_certificate(this, reader);
22             }
23             if (this->peer)
24             {
25                 expected = TLS_CERTIFICATE;
26                 break;
27             }
28             /* otherwise fall through to next state */
29         case STATE_CERT_RECEIVED:
30             if (type == TLS_CERTIFICATE_VERIFY)
31             {
32                 return process_cert_verify(this, reader);
33             }
34             if (this->peer)
35             {
36                 expected = TLS_CERTIFICATE_VERIFY;
37                 break;
38             }
39             /* otherwise fall through to next state */
40         case STATE_CERT_VERIFY_RECEIVED:
41             if (type == TLS_FINISHED)
42             {
43                 return process_finished(this, reader);
44             }
45             return NEED_MORE;
46         case STATE_FINISHED_RECEIVED:
47             if (type == TLS_KEY_UPDATE)
48             {
49                 return process_key_update(this, reader);
50             }
51             return INVALID_STATE;
52         default:
53             DBG1(DBG_TLS, "TLS %N not expected in current state",
54                 tls_handshake_type_names, type);
55             this->alert->add(this->alert, TLS_FATAL, TLS_UNEXPECTED_MESSAGE);
56             return NEED_MORE;
57     }
58 }

```

---

Listing 11: Server-side process state machine with mutual authentication support.

---

```

1  if (this->tls->get_version_max(this->tls) < TLS_1_3)
2  {
3      switch (this->state)
4      {
5          /* removed */
6      }
7      else
8      {
9          switch (this->state)
10         {
11             case STATE_HELLO_RECEIVED:
12                 return send_server_hello(this, type, writer);
13             case STATE_HELLO_SENT:
14             case STATE_CIPHERSPEC_CHANGED_OUT:
15                 return send_encrypted_extensions(this, type, writer);
16             case STATE_ENCRYPTED_EXTENSIONS_SENT:
17                 if (this->peer)
18                 {
19                     return send_certificate_request(this, type, writer);
20                 }
21                 /* otherwise fall through to next state */
22             case STATE_CERTREQ_SENT:
23                 return send_certificate(this, type, writer);
24             case STATE_CERT_SENT:
25                 return send_certificate_verify(this, type, writer);
26             case STATE_CERT_VERIFY_SENT:
27                 return send_finished(this, type, writer);
28             case STATE_FINISHED_SENT:
29                 if (!this->crypto->derive_app_keys(this->crypto))
30                 {
31                     this->alert->add(this->alert, TLS_FATAL, TLS_INTERNAL_ERROR);
32                     return NEED_MORE;
33                 }
34                 /* inbound key switches after process client finished message */
35                 this->crypto->change_cipher(this->crypto, FALSE);
36                 this->state = STATE_FINISHED_SENT_KEY_SWITCHED;
37                 return INVALID_STATE;
38             case STATE_KEY_UPDATE_REQUESTED:
39                 return send_key_update(this, type, writer);
40             case STATE_KEY_UPDATE_SENT:
41                 if (!this->crypto->update_app_keys(this->crypto, FALSE))
42                 {
43                     this->alert->add(this->alert, TLS_FATAL, TLS_INTERNAL_ERROR);
44                     return NEED_MORE;
45                 }
46                 this->crypto->change_cipher(this->crypto, FALSE);
47                 this->state = STATE_FINISHED_RECEIVED;
48             default:
49                 return INVALID_STATE;
50         }
51     }

```

---

Listing 12: Server-side build state machine with mutual authentication support.

In TLS 1.3 there is a static byte chunk that is used by both peers. It consists of 64 space characters, the string “TLS 1.3, client CertificateVerify” and a single zero byte delimiter.

We encapsulated this chunk of byte in `tls_crypto.c` because both peers need those chunks, see listing 13.

---

```
1  /**
2   * TLS 1.3 static part of the data the peer signs (64 spaces followed by the
3   * context string "TLS 1.3, client CertificateVerify" and a 0 byte).
4   */
5  static chunk_t tls13_sig_data_client = chunk_from_chars(
6      0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
7      0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
8      0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
9      0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
10     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
11     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
12     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
13     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
14     0x54, 0x4c, 0x53, 0x20, 0x31, 0x2e, 0x33, 0x2c,
15     0x20, 0x63, 0x6c, 0x69, 0x65, 0x6e, 0x74, 0x20,
16     0x43, 0x65, 0x72, 0x74, 0x69, 0x66, 0x69, 0x63,
17     0x61, 0x74, 0x65, 0x56, 0x65, 0x72, 0x69, 0x66,
18     0x79, 0x00,
19 );
```

---

Listing 13: Static blob to generate client certificate verify signature.

The second commit<sup>6</sup> addresses the client implementation of mutual authentication in TLS 1.3. Here, alongside with message processing and building code, the process state machine had to be adapted slightly to parse messages of type `TLS_CERTIFICATE_REQUEST`, as shown in listing 14. The build state machine also needed only a few enhancements to send `Certificate` and `CertificateVerify` messages, as of listing 15.

---

```
1  if (type == TLS_CERTIFICATE_REQUEST)
2  {
3      return process_certreq(this, reader);
4  }
5  /* no cert request, server does not want to authenticate us */
6  DESTROY_IF(this->peer);
7  this->peer = NULL;
8  /* otherwise fall through to next state */
9  case STATE_CERTREQ_RECEIVED:
```

---

Listing 14: Client-side process state machine with mutual authentication support.

---

<sup>6</sup>6849c0a1: tls-peer: Mutual authentication support for TLS 1.3

---

```

1  if (this->peer)
2  {
3      return send_certificate(this, type, writer);
4  }
5  /* otherwise fall through to next state */
6  case STATE_CERT_SENT:
7      if (this->peer)
8      {
9          return send_certificate_verify(this, type, writer);
10     }
11     /* otherwise fall through to next state */
12 case STATE_VERIFY_SENT:

```

---

Listing 15: Client-side build state machine with mutual authentication support.

### 5.3. PSK-based resumption

The PSK-based resumption feature was not completed within this bachelor thesis. However, the foundation of generating the necessary cryptographic keys has been laid. The HKDF implementation now provides the interface to generate the resumption secret at the end of the first session and the PSK binder used to initiate a subsequent session. The generated keys were verified by additional HKDF unit tests that were added to the test suite.

The two newly added methods `resume()` and `binder`, as shown in 5.1, provide the ability to generate the PSK secrets.

HKDF
<b>+ <code>tls_hkdf_create(hash_algorithm_t, chunk_t): tls_hkdf_t</code></b> <b>+ <code>set_shared_secret(tls_hkdf_t *, chunk_t *): void</code></b> <b>+ <code>generate_secret(tls_hkdf_t *, tls_hkdf_labels_t, chunk_t, chunk_t *): bool</code></b> <b>+ <code>derive_key(tls_hkdf_t *, bool, size_t, chunk_t *): bool</code></b> <b>+ <code>derive_iv(tls_hkdf_t *, bool, size_t, chunk_t *): bool</code></b> <b>+ <code>derive_finished(tls_hkdf_t *, bool, chunk_t *): bool</code></b> <b>+ <code>export(tls_hkdf_t *, char *, chunk_t, chunk_t, size_t, chunk_t *): bool</code></b> <b>+ <code>resume(tls_hkdf_t *, chunk_t, chunk_t, chunk_t *): bool</code></b> <b>+ <code>binder(tls_hkdf_t *, chunk_t, chunk_t *): bool</code></b> <b>+ <code>allocate_bytes(tls_hkdf_t *, chunk_t, chunk_t, chunk_t *): bool</code></b> <b>+ <code>destroy(tls_hkdf_t *): void</code></b>

Figure 5.1.: HKDF with new interface for PSK.

Listing 16 shows the newly added methods with the describing Doxygen documentation.

---

```

1  /**
2   * Generate resumption PSKs.
3   *
4   * @param messages      handshake messages
5   * @param nonce          nonce to use for this PSK
6   * @param psk            generated PSK
7   * @return              TRUE if PSK successfully generated
8   */
9  bool (*resume)(tls_hkdf_t *this, chunk_t messages, chunk_t nonce, chunk_t *psk);
10
11 /**
12  * Generate a PSK binder.
13  *
14  * @param seed            transcript-hash of client_hello to seed the PRF
15  * @param psk_binder      generated psk binder
16  * @return               TRUE if output was generated
17  */
18 bool (*binder)(tls_hkdf_t *this, chunk_t seed, chunk_t *psk_binder);

```

---

Listing 16: HKDF interface with resume() and binder().

The implementation of resume() was added in listing 17 and binder() in listing 18.

---

```

1  METHOD(tls_hkdf_t, resume, bool,
2      private_tls_hkdf_t *this, chunk_t messages, chunk_t nonce, chunk_t *key)
3  {
4      chunk_t resumption_master;
5
6      if (this->phase != HKDF_PHASE_3)
7      {
8          DBG1(DBG_TLS, "unable to generate resumption key material");
9          return FALSE;
10     }
11     if (!nonce.len)
12     {
13         DBG1(DBG_TLS, "no nonce provided");
14         return FALSE;
15     }
16
17     /**
18      * PSK associated with the ticket according to RFC 8446, section 4.6.1
19      *
20      * HKDF-Expand-Label(resumption_master_secret,
21      *                    "resumption", ticket_nonce, Hash.length)
22      */
23     if (!generate_secret(this, TLS_HKDF_RES_MASTER, messages,
24                         &resumption_master))
25     {
26         DBG1(DBG_TLS, "unable to derive resumption master secret");
27         return FALSE;
28     }
29
30     if (!expand_label(this, resumption_master, chunk_from_str("resumption"),
31                     nonce, this->hasher->get_hash_size(this->hasher), key))
32     {
33         chunk_free(&resumption_master);
34         DBG1(DBG_TLS, "unable to expand key material");
35         return FALSE;
36     }
37     chunk_free(&resumption_master);
38     return TRUE;
39 }

```

---

Listing 17: HKDF resume() implementation.

---

```

1  METHOD(tls_hkdf_t, binder, bool,
2      private_tls_hkdf_t *this, chunk_t seed, chunk_t *out)
3  {
4      chunk_t binder_key, finished_key;
5
6      if (!generate_secret(this, TLS_HKDF_RES_BINDER, chunk_empty, &binder_key))
7      {
8          DBG1(DBG_TLS, "unable to derive binder key");
9          return FALSE;
10     }
11
12     if (!expand_label(this, binder_key, chunk_from_str("finished"), chunk_empty,
13         this->hasher->get_hash_size(this->hasher), &finished_key))
14     {
15         chunk_free(&binder_key);
16         return FALSE;
17     }
18     chunk_free(&binder_key);
19
20     if (!this->prf->set_key(this->prf, finished_key) ||
21         !this->prf->allocate_bytes(this->prf, seed, out))
22     {
23         chunk_free(&finished_key);
24         return FALSE;
25     }
26     chunk_free(&finished_key);
27     return TRUE;
28 }

```

---

Listing 18: HKDF binder() implementation.

As described in 4.3, it is crucial that information from a previous session persists to a subsequent session. strongSwan's libtls has already implemented such a persistency feature in the cache module (tls\_cache.h and tls\_cache.c). However, it was designed for use in legacy TLS versions. Also, the cache module, from the peers' perspective, is encapsulated entirely in the crypto module (tls\_crypto.h and tls\_crypto.c). For a final PSK implementation, the steps described in 5.3.1 are suggested.

### 5.3.1. Next steps

For a final PSK-based resumption implementation, the following steps would be needed:

1. Key-value oriented cache module can be reused untouched, when the *session ticket* is used as searchable key and the PSK session secret (generated with the nonce) is concatenated with the *age add* and stored as value.
2. The crypto module needs additional interfaces to allow peers to add and receive TLS 1.3 related resumption information.
3. Client and server need additional message building and processing logic as well as a review of the state machine when a PSK-based resumption is required. It is important that the HKDF object initialised with the PSK is not reinitialised without the PSK as it was used in the past.



## 5.4. Unit Tests

During this thesis, multiple new unit test cases were added, as a comparison of figure 5.2 to figure 5.3 shows.

```

Running 3 'libtls' test suites:
Running suite 'HKDF TLS 1.3':
Running case 'Ulfheim Keys': ++
Running case 'RFC 8448': ++
Passed all 2 'HKDF TLS 1.3' test cases
Running suite 'socket':
Running case 'TLS [1.0..1.3] client to TLS 1.3 server': ++++
Running case 'TLS 1.3 client to TLS [1.0..1.3] server': ++++
Running case 'TLS [1.0..1.3] client to TLS 1.2 server': ++++
Running case 'TLS 1.3/curves': ++++++
Running case 'TLS 1.3/anon': +++
Running case 'TLS 1.3/mutl': +++
Running case 'TLS 1.2/anon': ++++++
Running case 'TLS 1.2/mutl': ++++++
Running case 'TLS 1.1/anon': ++++++
Running case 'TLS 1.1/mutl': ++++++
Running case 'TLS 1.0/anon': ++++++
Running case 'TLS 1.0/mutl': ++++++
Running case 'TLS 1.0/ed25519': +++
Running case 'TLS 1.2/ed25519': ++++++
Running case 'TLS 1.1/ed25519': ++++++
Running case 'TLS 1.0/ed25519': ++++++
Running case 'TLS 1.3/ed448': +++
Running case 'TLS 1.2/ed448': ++++++
Running case 'TLS 1.1/ed448': ++++++
Running case 'TLS 1.0/ed448': ++++++
Passed all 20 'socket' test cases
Running suite 'suites':
Running case 'cipher-names': +
Passed all 1 'suites' test cases
Passed all 3 'libtls' suites

```

Figure 5.2.: Unit tests before this bachelor thesis.

Figure 5.3.: Unit tests after this bachelor thesis.

Here, the newly added test cases are discussed:

'RFC 8448' : The 'RFC 8448' test suite provides already two test cases for testing the HKDF implementation. These test cases cover a simple TLS 1.3 handshake provided on keys and handshake data by <https://tls13.ulfheim.net/>. There also exists RFC 8448<sup>7</sup> which provides handshake data and keys for a broad set of TLS 1.3 handshake use cases. Here, additional tests were added that cover a larger set of functions of the implemented HKDF especially for 0-RTT and 1-RTT handshake scenarios.

'TLS [1.0..1.3] client to TLS 1.3 server' : Test client to server TLS connection where the client tries each TLS version from 1.0, 1.1, 1.2 and 1.3 to a TLS server has been fixed to TLS 1.3.

'TLS 1.3 client to TLS [1.0..1.3] server' : Test client fixed to TLS 1.3 to connect to a server with support for TLS 1.0, 1.1, 1.2 or 1.3.

<sup>7</sup>24.

- 'TLS [1.0..1.3] client to TLS 1.2 server' : Test client to server TLS connection where the client tries each TLS version from 1.0, 1.1, 1.2 and 1.3 to a TLS server fixed to TLS 1.2.
- 'TLS 1.3/curves' : Tests all supported TLD DH groups in a TLS 1.3 connection between client and server.
- 'TLS 1.3/anon' : Test TLS 1.3 connection from client to server without client authentication.
- 'TLS 1.3/mutl1' : Test TLS 1.3 connection from client to server with client authentication.
- 'TLS 1.3/ed25519' : Test client to server TLS connection with either an RSA or an Ed25519 key with all supported TLS 1.3 cipher suites.
- 'TLS 1.2/ed25519' : Test client to server TLS connection with either an RSA or an Ed25519 key with all supported TLS 1.2 cipher suites.
- 'TLS 1.1/ed25519' : Test client to server TLS connection with either an RSA or an Ed25519 key with all supported TLS 1.1 cipher suites.
- 'TLS 1.0/ed25519' : Test client to server TLS connection with either an RSA or an Ed25519 key with all supported TLS 1.0 cipher suites.
- 'TLS 1.3/ed448' : Test client to server TLS connection with either an RSA or an Ed448 key with all supported TLS 1.3 cipher suites.
- 'TLS 1.2/ed448' : Test client to server TLS connection with either an RSA or an Ed448 key with all supported TLS 1.2 cipher suites.
- 'TLS 1.1/ed448' : Test client to server TLS connection with either an RSA or an Ed448 key with all supported TLS 1.1 cipher suites.
- 'TLS 1.0/ed448' : Test client to server TLS connection with either an RSA or an Ed448 key with all supported TLS 1.0 cipher suites.

Before, libt1s had eight test cases and 165 test asserts (indicated as green '+' character) in three test suites. Now it provides 23 test cases and 300 test asserts. This makes a plus of 15 test cases and 135 test asserts within the libt1s test suites.

These newly added unit tests slightly increase the coverage of libt1s:



Figure 5.4.: Unit test coverage of libt1s before this thesis (lines/functions/branches).

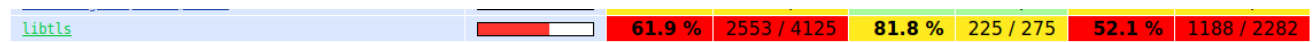


Figure 5.5.: Unit test coverage of libt1s after this thesis (lines/functions/branches).

Line coverage increased by 7.7% to 61.9%, function coverage by 7.4% to 81.8% and branch coverage by 9.8% to 52.1%. The detailed file comparison follows:














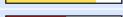




Filename	Line Coverage ↕	Functions ↕	Branches ↕
tls.c	 59.6 % 90 / 151	60.0 % 9 / 15	44.3 % 27 / 61
tls_aead.c	 80.3 % 57 / 71	100.0 % 8 / 8	50.0 % 10 / 20
tls_aead_expl.c	 84.2 % 64 / 76	100.0 % 8 / 8	52.2 % 24 / 46
tls_aead_impl.c	 85.1 % 63 / 74	100.0 % 8 / 8	52.1 % 25 / 48
tls_aead_null.c	 87.0 % 40 / 46	100.0 % 8 / 8	50.0 % 8 / 16
tls_aead_seq.c	 0.0 % 0 / 93	0.0 % 0 / 9	0.0 % 0 / 35
tls_alert.c	 65.2 % 30 / 46	100.0 % 6 / 6	47.1 % 8 / 17
tls_cache.c	 0.0 % 0 / 63	0.0 % 0 / 8	0.0 % 0 / 38
tls_compression.c	 100.0 % 10 / 10	100.0 % 4 / 4	- 0 / 0
tls_crypto.c	 60.0 % 403 / 672	78.7 % 37 / 47	45.1 % 213 / 472
tls_eap.c	 0.0 % 0 / 189	0.0 % 0 / 11	0.0 % 0 / 109
tls_fragmentation.c	 77.7 % 160 / 206	91.7 % 11 / 12	66.3 % 69 / 104
tls_hkdf.c	 52.3 % 123 / 235	88.2 % 15 / 17	38.9 % 49 / 126
tls_peer.c	 44.9 % 400 / 891	75.0 % 27 / 36	36.5 % 169 / 463
tls_prf.c	 87.3 % 48 / 55	100.0 % 9 / 9	53.8 % 14 / 26
tls_protection.c	 80.0 % 32 / 40	100.0 % 6 / 6	68.2 % 15 / 22
tls_server.c	 67.7 % 321 / 474	85.2 % 23 / 27	59.3 % 137 / 231
tls_socket.c	 52.6 % 91 / 173	63.6 % 7 / 11	51.0 % 49 / 96

Figure 5.6.: Unit test coverage of libtls before this thesis in detail.










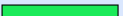


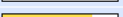





Filename	Line Coverage ↕	Functions ↕	Branches ↕
tls.c	 60.9 % 92 / 151	60.0 % 9 / 15	49.2 % 30 / 61
tls_aead.c	 80.3 % 57 / 71	100.0 % 8 / 8	50.0 % 10 / 20
tls_aead_expl.c	 84.2 % 64 / 76	100.0 % 8 / 8	52.2 % 24 / 46
tls_aead_impl.c	 85.1 % 63 / 74	100.0 % 8 / 8	52.1 % 25 / 48
tls_aead_null.c	 87.0 % 40 / 46	100.0 % 8 / 8	50.0 % 8 / 16
tls_aead_seq.c	 77.4 % 72 / 93	88.9 % 8 / 9	48.6 % 17 / 35
tls_alert.c	 65.2 % 30 / 46	100.0 % 6 / 6	47.1 % 8 / 17
tls_cache.c	 0.0 % 0 / 63	0.0 % 0 / 8	0.0 % 0 / 38
tls_compression.c	 100.0 % 10 / 10	100.0 % 4 / 4	- 0 / 0
tls_crypto.c	 70.9 % 583 / 822	91.8 % 56 / 61	55.9 % 320 / 572
tls_eap.c	 0.0 % 0 / 189	0.0 % 0 / 11	0.0 % 0 / 109
tls_fragmentation.c	 77.7 % 160 / 206	91.7 % 11 / 12	65.4 % 68 / 104
tls_hkdf.c	 58.8 % 163 / 277	95.0 % 19 / 20	49.3 % 73 / 148
tls_peer.c	 56.0 % 519 / 927	80.6 % 29 / 36	53.2 % 259 / 487
tls_prf.c	 87.3 % 48 / 55	100.0 % 9 / 9	53.8 % 14 / 26
tls_protection.c	 84.1 % 37 / 44	100.0 % 6 / 6	83.3 % 25 / 30
tls_server.c	 65.3 % 524 / 802	82.9 % 29 / 35	60.1 % 258 / 429
tls_socket.c	 52.6 % 91 / 173	63.6 % 7 / 11	51.0 % 49 / 96

Figure 5.7.: Unit test coverage of libtls after this thesis in detail.

How the coverage was measured is described in C.1.1.

## 6. Testing

This chapter describes how the implementation in `libtls` is tested throughout this thesis. First, client functionality was tested, followed by server-side and mutual authentication functionality.

For this manual testing and functionality proofing, the strongSwan tool `tls_test` was used. The compilation steps described in appendix C also built additional executables. `tls_test` was built, among various other programmes, in the “scripts” directory. This executable provides TLS client and server functionality similar to OpenSSL’s “s\_client” and “s\_server”.

This CLI tool provides several parameters that were used to manually test client and server functionality. Listing 19 shows all parameters and their description.

---

```
$ ./scripts/tls_test --help
usage:
/home/pascal/Documents/Bildung/fh/hsr/sem7/BA/build/scripts/.libs/tls_test --connect
↪ <address> --port <port> [--key <key>] [--cert <file>] [--cacert <file>]+ [--times
↪ <n>]
/home/pascal/Documents/Bildung/fh/hsr/sem7/BA/build/scripts/.libs/tls_test --listen
↪ <address> --port <port> --key <key> --cert <file> [--cacert <file>]+ [--times <n>]

options:
--help                print help and exit
--connect <address>   connect to a server on dns name or ip address
--listen <address>    listen on dns name or ip address
--port <port>         specify the port to use
--cert <file>         certificate to authenticate itself
--key <file>          private key to authenticate itself
--cacert <file>       certificate to verify other peer
--times <n>           specify the amount of repeated connection establishments
--ipv4                use IPv4
--ipv6                use IPv6
--min-version <version> specify the minimum TLS version, supported versions:
                        1.0 (default), 1.1, 1.2 and 1.3
--max-version <version> specify the maximum TLS version, supported versions:
                        1.0, 1.1, 1.2 and 1.3 (default)
--version <version>   set one specific TLS version to use, supported versions:
                        1.0, 1.1, 1.2 and 1.3
--debug <debug level> set debug level, default is 1
```

---

Listing 19: Help output of `tls_test` utility tool.

For space reasons, only the output of `tls_test` is listed here. All OpenSSL commands used here can be found in appendix E. The used self-signed RSA 2048 bit X.509 certificate and its private key are included in F. All used Wireshark packet captures – and some additional ones

from appendix E – with included decryption keys<sup>1</sup> are provided on my web server <https://www.addere.ch/ba/wireshark/>.

## 6.1. Client

This section focuses on client-side test scenarios.

### 6.1.1. `tls_test` with TLS 1.3 to OpenSSL server

Here, `tls_test` restricted to require TLS 1.3 was used to establish a secured connection to an OpenSSL server.

OpenSSL server:

```
"openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
-keyform der -key caKey.der -certform der -cert cert.der"
```

---

```
$ ./scripts/tls_test --connect localhost --port 8443 --key caKey.der --cert cert.der
```

```
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
received TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
  using trusted certificate "C=CH, O=strongSwan, CN=localhost"
ping
pong
~C
```

---

Listing 20: `tls_test` connection to OpenSSL using TLS 1.3.

Protocol	Length	Info
TLSv1.3	394	Client Hello
TLSv1.3	1417	Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate, Certificate Verify, Finished
TLSv1.3	140	Finished
TLSv1.3	321	New Session Ticket
TLSv1.3	321	New Session Ticket
TLSv1.3	89	Application Data
TLSv1.3	93	Application Data
TLSv1.3	93	Application Data

Figure 6.1.: `tls_test` with TLS 1.3 to OpenSSL server.

### 6.1.2. `tls_test` with TLS 1.2 to OpenSSL server

Here, `tls_test` restricted to require TLS 1.2 was used to establish a secured connection to an OpenSSL server.

OpenSSL server:

---

<sup>1</sup>I would like to take this opportunity to thank Peter Wu for providing the very useful tool “inject-tls-secrets.py” <https://gist.github.com/Lekensteyn/f64ba6d6d2c6229d6ec444647979ea24> visited 2020-12-10.

```
"openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
-keyform der -key caKey.der -certform der -cert cert.der"
```

---

```
$ ./scripts/tls_test --connect localhost --port 8443 --key caKey.der --cert cert.der
↪ --version 1.2
```

```
negotiated TLS 1.2 using suite TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
received TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
  using trusted certificate "C=CH, O=strongSwan, CN=localhost"
ping
pong
^C
```

---

Listing 21: `tls_test` limited to TLS 1.2 connection to OpenSSL.

Protocol	Length	Info
TLSv1.2	300	Client Hello
TLSv1.2	1295	Server Hello, Certificate, Server Key Exchange, Server Hello Done
TLSv1.2	192	Client Key Exchange, Change Cipher Spec, Finished
TLSv1.2	117	Change Cipher Spec, Finished
TLSv1.2	96	Application Data
TLSv1.2	100	Application Data
TLSv1.2	100	Application Data

Figure 6.2.: `tls_test` with TLS 1.2 to OpenSSL server.

### 6.1.3. `tls_test` to Google server on IPv4

Here, `tls_test` without restrictions was used to connect to a Google server. TLS 1.3 was negotiated.

---

```
./scripts/tls_test --connect www.google.com --port 443 --debug 1 --ipv4 --cert
↪ /usr/share/ca-certificates/mozilla/GlobalSign_Root_CA_-_R2.crt

negotiated TLS 1.3 using suite TLS_AES_128_GCM_SHA256
server requests key exchange with CURVE25519
negotiated TLS 1.3 using suite TLS_AES_128_GCM_SHA256
received TLS server certificate 'C=US, ST=California, L=Mountain View, O=Google LLC,
↪ CN=www.google.com'
received TLS intermediate certificate 'C=US, O=Google Trust Services, CN=GTS CA 101'
  using certificate "C=US, ST=California, L=Mountain View, O=Google LLC,
  ↪ CN=www.google.com"
  using untrusted intermediate certificate "C=US, O=Google Trust Services, CN=GTS CA
  ↪ 101"
  using trusted ca certificate "OU=GlobalSign Root CA - R2, O=GlobalSign, CN=GlobalSign"
  reached self-signed root ca with a path length of 1
^C
```

---

Listing 22: TLS connection to Google on IP v4 and port 443/tcp using `tls_test`.

Establish a TLS 1.3 connection to the host “`www.google.com`” on port “443” over IP version 4. Use the certificate “`/usr/share/ca-certificates/mozilla/GlobalSign_Root_CA_-_R2.crt`” to

verify the TLS authentication. Use debugging level 1 instead of -1.

#### 6.1.4. `tls_test` to Google server on IPv6

Here, `tls_test` without restrictions was used to connect to a Google server. TLS 1.3 was negotiated.

---

```
$ ./scripts/tls_test --connect www.google.com --port 443 --debug 1 --ipv6 --cert
↪ /usr/share/ca-certificates/mozilla/GlobalSign_Root_CA_-_R2.crt --max-version 1.2

negotiated TLS 1.2 using suite TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
received TLS server certificate 'C=US, ST=California, L=Mountain View, O=Google LLC,
↪ CN=www.google.com'
received TLS intermediate certificate 'C=US, O=Google Trust Services, CN=GTS CA 101'
  using certificate "C=US, ST=California, L=Mountain View, O=Google LLC,
  ↪ CN=www.google.com"
  using untrusted intermediate certificate "C=US, O=Google Trust Services, CN=GTS CA
  ↪ 101"
  using trusted ca certificate "OU=GlobalSign Root CA - R2, O=GlobalSign, CN=GlobalSign"
  reached self-signed root ca with a path length of 1
^C
```

---

Listing 23: TLS connection to Google on IP v6 and port 443/tcp using `tls_test`.

Establish a TLS 1.2 connection to the host “www.google.com” on port “443” over IP version 6. Use the certificate “/usr/share/ca-certificates/mozilla/GlobalSign\_Root\_CA\_-\_R2.crt” to verify the TLS authentication. Use debugging level 1 instead of -1.

#### 6.1.5. `tls_test` to `tls13.id-pw` server

Here, `tls_test` without restrictions was used to connect to the `tls13.id.pw`<sup>2</sup> server. TLS 1.3 was negotiated.

---

<sup>2</sup>This is a good TLS client testing website because it randomly uses different TLS configuration on each connection.

---

```
$ (echo -e "GET / HTTP/1.1\nHost: tls13.1d.pw\n"; cat) | scripts/tls_test --connect
↪ tls13.1d.pw --port 443 --cert
↪ /usr/share/ca-certificates/mozilla/Comodo_AAA_Services_root.crt --times 1

negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
server requests key exchange with CURVE25519
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
received TLS server certificate 'CN=tls13.1d.pw'
received TLS intermediate certificate 'C=GB, ST=Greater Manchester, L=Salford, O=Sectigo
↪ Limited, CN=Sectigo ECC Domain Validation Secure Server CA'
received TLS intermediate certificate 'C=US, ST=New Jersey, L=Jersey City, O=The
↪ USERTRUST Network, CN=USERTrust ECC Certification Authority'
using certificate "CN=tls13.1d.pw"
using untrusted intermediate certificate "C=GB, ST=Greater Manchester, L=Salford,
↪ O=Sectigo Limited, CN=Sectigo ECC Domain Validation Secure Server CA"
using untrusted intermediate certificate "C=US, ST=New Jersey, L=Jersey City, O=The
↪ USERTRUST Network, CN=USERTrust ECC Certification Authority"
using trusted ca certificate "C=GB, ST=Greater Manchester, L=Salford, O=Comodo CA
↪ Limited, CN=AAA Certificate Services"
reached self-signed root ca with a path length of 2
received TLS close notify
sending TLS close notify
...
```

---

Listing 24: TLS connection to tls13.1d.pw on port 443/tcp using tls\_test.

Establish a TLS 1.3 connection to the host “tls13.1d.pw” on port “443”. Use the certificate “/usr/share/ca-certificates/mozilla/Comodo\_AAA\_Services\_root.crt” to verify the TLS authentication. Do not connect multiple times but only once. Once connected send the payload “GET / HTTP/1.1\nHost: tls13.1d.pw\n”.

## 6.2. Server

This section focuses on server-side test scenarios.

### 6.2.1. OpenSSL to tls\_test

Here, OpenSSL without any version restriction was used to connect to tls\_test. TLS 1.3 was negotiated.

OpenSSL client:

```
“openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys”
```



---

```
$ ./scripts/tls_test --listen localhost --port 8443 --key caKey.der --cert cert.der
127.0.0.1[53872] connected
using key of type RSA
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
using key exchange CURVE25519
sending TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
ping
pong
127.0.0.1[53872] disconnected
sending TLS close notify
^C
```

---

Listing 25: OpenSSL connection to `tls_test` using TLS 1.3.

Protocol	Length	Info
TLSv1.3	349	Client Hello
TLSv1.3	1334	Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate, Certificate Verify, Finished
TLSv1.3	146	Change Cipher Spec, Finished
TLSv1.3	93	Application Data
TLSv1.3	93	Application Data

Figure 6.3.: OpenSSL to `tls_test`.

### 6.2.2. OpenSSL 1.2 to `tls_test`

Here, OpenSSL restricted to TLS 1.2 was used to connect to `tls_test`. TLS 1.2 was negotiated.

OpenSSL client:

```
"openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
-tls1_2"
```

---

```
$ ./scripts/tls_test --listen localhost --port 8443 --key caKey.der --cert cert.der
127.0.0.1[53902] connected
using key of type RSA
negotiated TLS 1.2 using suite TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
sending TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
ping
pong
127.0.0.1[53902] disconnected
sending TLS close notify
^C
```

---

Listing 26: OpenSSL limited to TLS 1.2 connection to `tls_test`.

Protocol	Length	Info
TLSv1.2	254	Client Hello
TLSv1.2	1286	Server Hello, Certificate, Server Key Exchange, Server Hello Done
TLSv1.2	224	Client Key Exchange, Change Cipher Spec, Finished
TLSv1.2	117	Change Cipher Spec, Finished
TLSv1.2	100	Application Data
TLSv1.2	100	Application Data

Figure 6.4.: OpenSSL 1.2 to `tls_test`.

### 6.2.3. OpenSSL to `tls_test` TLS 1.2

Here, OpenSSL without any version restriction was used to connect to `tls_test` restricted to TLS 1.2. TLS 1.2 was negotiated.

OpenSSL client:

```
"openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys"
```

---

```
$ ./scripts/tls_test --listen localhost --port 8443 --key caKey.der --cert cert.der
↳ --version 1.2
127.0.0.1[53920] connected
using key of type RSA
negotiated TLS 1.2 using suite TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
sending TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
ping
pong
127.0.0.1[53920] disconnected
sending TLS close notify
^C
```

---

Listing 27: OpenSSL connection to `tls_test` limited TLS 1.2.

Protocol	Length	Info
TLSv1.2	349	Client Hello
TLSv1.2	1286	Server Hello, Certificate, Server Key Exchange, Server Hello Done
TLSv1.2	224	Client Key Exchange, Change Cipher Spec, Finished
TLSv1.2	117	Change Cipher Spec, Finished
TLSv1.2	100	Application Data
TLSv1.2	100	Application Data

Figure 6.5.: OpenSSL to `tls_test` TLS 1.2.

### 6.2.4. `tls_test` to `tls_test`

Here, `tls_test` without any version restriction was used to connect to `tls_test`. TLS 1.3 was negotiated.

---

```
$ ./scripts/tls_test --connect localhost --port 8443 --key caKey.der --cert cert.der

negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
received TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
  using trusted certificate "C=CH, O=strongSwan, CN=localhost"
ping
pong
^C
```

---

Listing 28: tls\_test connection to tls\_test using TLS 1.3 client side.

---

```
$ ./scripts/tls_test --listen localhost --port 8443 --key caKey.der --cert cert.der
127.0.0.1[53944] connected
using key of type RSA
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
using key exchange SECP256R1
sending TLS server certificate 'C=CH, O=strongSwan, CN=localhost'

ping
pong
127.0.0.1[53944] disconnected
sending TLS close notify
^C
```

---

Listing 29: tls\_test connection to tls\_test using TLS 1.3 server side.

Protocol	Length	Info
TLSv1.3	394	Client Hello
TLSv1.3	1335	Server Hello, Change Cipher Spec, Application Data
TLSv1.3	140	Application Data
TLSv1.3	89	Application Data
TLSv1.3	93	Application Data
TLSv1.3	93	Application Data

Figure 6.6.: tls\_test to tls\_test.

### 6.2.5. KeyUpdate

libtls does not actively request key updates to its peers but it supports the messages and key switching and it answers requests for updates. The tests in this section are accomplished with OpenSSL as peer because it supports key update requests as described in E.4.

Both tests follow this test procedure:

1. Connection establishment
2. Client sends “ping”
3. Server sends “pong”
4. OpenSSL peer sends a key update message to tls\_test peer “k”
5. Client sends “ping ping”

6. Client sends “pong pong”
7. OpenSSL peer sends a key update message and a key update request to `tls_test` peer “K”
8. Client sends “ping ping ping”
9. Client sends “pong pong pong”

A connection from `tls_test` client to an OpenSSL server is listed in 30, where the server updates its key first and then asks for a key update by the peer<sup>3</sup> once the connection is established.

OpenSSL server:

```
“openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
-keyform der -key caKey.der -certform der -cert cert.der”
```

---

```
$ ./scripts/tls_test --connect localhost --port 8443 --key caKey.der --cert cert.der
↪ --debug 2
connecting to 127.0.0.1[8443]
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
received TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
  using trusted certificate "C=CH, O=strongSwan, CN=localhost"
ping
sending TLS ApplicationData record (22 bytes)
processing TLS ApplicationData record (22 bytes)
pong
processing TLS ApplicationData record (22 bytes)
received TLS KeyUpdate handshake (1 bytes)
ping ping
sending TLS ApplicationData record (27 bytes)
processing TLS ApplicationData record (27 bytes)
pong pong
processing TLS ApplicationData record (22 bytes)
received TLS KeyUpdate handshake (1 bytes)
server requested KeyUpdate
sending TLS KeyUpdate handshake (1 bytes)
sending TLS ApplicationData record (22 bytes)
ping ping ping
sending TLS ApplicationData record (32 bytes)
processing TLS ApplicationData record (32 bytes)
pong pong pong
^C
```

---

Listing 30: Key update message and key update request from the OpenSSL server.

---

<sup>3</sup>See also appendix E.4

Protocol	Length	Info
TLSv1.3	394	Client Hello
TLSv1.3	1417	Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate, Certificate Verify, Finished
TLSv1.3	140	Finished
TLSv1.3	321	New Session Ticket
TLSv1.3	305	New Session Ticket
TLSv1.3	89	Application Data
TLSv1.3	93	Application Data
TLSv1.3	93	Application Data
TLSv1.3	93	Key Update
TLSv1.3	98	Application Data
TLSv1.3	98	Application Data
TLSv1.3	93	Key Update
TLSv1.3	93	Key Update
TLSv1.3	103	Application Data
TLSv1.3	103	Application Data

Figure 6.7.: KeyUpdate with OpenSSL server.

OpenSSL client connects to a libtls server. The client updates its own key first and then asks for a key update as of listing 31.

OpenSSL client:

```
"$ openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys"
```

---

```
$ ./scripts/tls_test --listen localhost --port 8443 --key caKey.der --cert cert.der
↪ --debug 2
127.0.0.1[52432] connected
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
ping
pong
sending TLS ApplicationData record (22 bytes)
processing TLS ApplicationData record (22 bytes)
received TLS KeyUpdate handshake (1 bytes)
processing TLS ApplicationData record (27 bytes)
ping ping
pong pong
sending TLS ApplicationData record (27 bytes)
processing TLS ApplicationData record (22 bytes)
received TLS KeyUpdate handshake (1 bytes)
client requested KeyUpdate
sending TLS KeyUpdate handshake (1 bytes)
sending TLS ApplicationData record (22 bytes)
processing TLS ApplicationData record (32 bytes)
ping ping ping
pong pong pong
sending TLS ApplicationData record (32 bytes)
127.0.0.1[52432] disconnected
sending TLS close notify
sending TLS ApplicationData record (19 bytes)
^C
```

---

Listing 31: Key update message and key update request from the OpenSSL client.

Protocol	Length	Info
TLSv1.3	349	Client Hello
TLSv1.3	1334	Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate, Certificate Verify, Finished
TLSv1.3	146	Change Cipher Spec, Finished
TLSv1.3	93	Application Data
TLSv1.3	93	Application Data
TLSv1.3	93	Key Update
TLSv1.3	98	Application Data
TLSv1.3	98	Application Data
TLSv1.3	93	Key Update
TLSv1.3	93	Key Update
TLSv1.3	103	Application Data
TLSv1.3	103	Application Data

Figure 6.8.: KeyUpdate with OpenSSL client.

In order to save space, both outputs were shortened.

## 6.2.6. HRR

First, a connection from `tls_test` client to an OpenSSL server is listed in 32 where the server requests a key share of X but the client sent Y, so a `HelloRetryRequest` was returned by the server.

OpenSSL server:

```
"openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
-keyform der -key caKey.der -certform der -cert cert.der --curves secp521r1"
```

---

```
$ ./scripts/tls_test --connect localhost --port 8443 --key caKey.der --cert cert.der
```

```
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
server requests key exchange with SECP521R1
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
received TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
  using trusted certificate "C=CH, O=strongSwan, CN=localhost"
ping
pong
^C
```

---

Listing 32: HRR message by OpenSSL server.

Protocol	Length	Info
TLSv1.3	394	Client Hello
TLSv1.3	133	Hello Retry Request, Change Cipher Spec
TLSv1.3	438	Client Hello
TLSv1.3	1463	Server Hello, Encrypted Extensions, Certificate, Certificate Verify, Finished
TLSv1.3	140	Finished
TLSv1.3	89	Application Data
TLSv1.3	305	New Session Ticket
TLSv1.3	321	New Session Ticket
TLSv1.3	93	Application Data
TLSv1.3	93	Application Data

Figure 6.9.: HRR with OpenSSL server.

Second, an OpenSSL client connects to `libtls` server. The client sends a key share of `sect233k1` and provides `sect233k1 (0x0006)` and `secp521r1 (0x0019)` as `supported_groups`

extension. The server supports the latter and therefore sent a HelloRetryRequest in return 33.

OpenSSL client:

```
"openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
--curves sect233k1:secp521r1"
```

---

```
$ ./scripts/tls_test --listen localhost --port 8443 --key caKey.der --cert cert.der
127.0.0.1[46148] connected
using key of type RSA
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
requesting key exchange with SECP521R1
resumed TLS 1.3 using suite TLS_AES_256_GCM_SHA384
using key exchange SECP521R1
sending TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
ping
pong
127.0.0.1[46148] disconnected
sending TLS close notify
```

---

Listing 33: HRR message by tls\_test server.

Protocol	Length	Info
TLSv1.3	372	Client Hello
TLSv1.3	159	Hello Retry Request
TLSv1.3	450	Change Cipher Spec, Client Hello
TLSv1.3	1435	Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate, Certificate Verify, Finished
TLSv1.3	140	Finished
TLSv1.3	93	Application Data
TLSv1.3	93	Application Data

Figure 6.10.: HRR with OpenSSL client.

If the client provides only unsupported DH groups, the server terminates the connection, as of listing 34.

OpenSSL client:

```
"openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
--curves sect233k1"
```

---

```
$ ./scripts/tls_test --listen localhost --port 8443 --key caKey.der --cert cert.der
127.0.0.1[46022] connected
using key of type RSA
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
no mutual supported group in client hello
sending fatal TLS alert 'illegal parameter'
127.0.0.1[46022] disconnected
```

---

Listing 34: Connection termination by server due to missing supported groups in client hello.

Protocol	Length	Info
TLSv1.2	370	Client Hello
TLSv1.2	73	Alert (Level: Fatal, Description: Illegal Parameter)

Figure 6.11.: HRR with OpenSSL server.

## 6.3. Mutual Authentication

Four test scenarios are described here. The first three tests are all permutations of a combination of OpenSSL and `tls_test`. In the last scenario, the OpenSSL client does not provide a certificate and the server terminates the connection.

### 6.3.1. OpenSSL to `tls_test`

OpenSSL client:

```
"openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
-keyform DER -key caKey.der -certform DER -cert cert.der"
```

---

```
$ ./scripts/tls_test --listen localhost --port 8443 --key caKey.der --cert cert.der
↪ --cacert cert.der
127.0.0.1[53514] connected
using key of type RSA
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
using key exchange CURVE25519
sending TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
received TLS peer certificate 'C=CH, O=strongSwan, CN=localhost'
    using trusted certificate "C=CH, O=strongSwan, CN=localhost"
ping
pong
127.0.0.1[53514] disconnected
sending TLS close notify
^C
```

---

Listing 35: Mutual authentication with OpenSSL to `tls_test`.

Protocol	Length	Info
TLSv1.3	349	Client Hello
TLSv1.3	1381	Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate Request, Certificate, Certificate Verify, Finished
TLSv1.3	1245	Change Cipher Spec, Certificate, Certificate Verify, Finished
TLSv1.3	93	Application Data
TLSv1.3	93	Application Data

Figure 6.12.: OpenSSL to `tls_test`.

### 6.3.2. `tls_test` to OpenSSL

OpenSSL server:

```
"openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
--keyform der --key caKey.der --certform der --cert cert.der --verify 1"
```



---

```
$ ./scripts/tls_test --connect localhost --port 8443 --key caKey.der --cert cert.der
↪ --cacert cert.der

negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
received TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
  using trusted certificate "C=CH, O=strongSwan, CN=localhost"
sending TLS client certificate 'C=CH, O=strongSwan, CN=localhost'
ping
pong
^C
```

---

Listing 36: Mutual authentication with `tls_test` to OpenSSL.

Protocol	Length	Info
TLSv1.3	394	Client Hello
TLSv1.3	1484	Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate Request, Certificate, Certificate Verify, Finished
TLSv1.3	1195	Certificate, Certificate Verify, Finished
TLSv1.3	89	Application Data
TLSv1.3	1105	New Session Ticket
TLSv1.3	1105	New Session Ticket
TLSv1.3	93	Application Data
TLSv1.3	93	Application Data

Figure 6.13.: `tls_test` to OpenSSL.

### 6.3.3. `tls_test` to `tls_test`

---

```
$ ./scripts/tls_test --connect localhost --port 8443 --key caKey.der --cert cert.der
↪ --cacert cert.der

negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
received TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
  using trusted certificate "C=CH, O=strongSwan, CN=localhost"
sending TLS client certificate 'C=CH, O=strongSwan, CN=localhost'
ping
pong
^C
```

---

Listing 37: Mutual authentication with `tls_test` to `tls_test` client side.

---

```

$ ./scripts/tls_test --listen localhost --port 8443 --key caKey.der --cert cert.der
↪ --cacert cert.der
127.0.0.1[53560] connected
using key of type RSA
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
using key exchange SECP256R1
sending TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
received TLS peer certificate 'C=CH, O=strongSwan, CN=localhost'
    using trusted certificate "C=CH, O=strongSwan, CN=localhost"

ping
pong
127.0.0.1[53560] disconnected
sending TLS close notify
^C

```

---

Listing 38: Mutual authentication with `tls_test` to `tls_test` server side.

Protocol	Length	Info
TLSv1.3	394	Client Hello
TLSv1.3	1382	Server Hello, Change Cipher Spec, Application Data
TLSv1.3	1195	Application Data
TLSv1.3	89	Application Data
TLSv1.3	93	Application Data
TLSv1.3	93	Application Data

Figure 6.14.: `tls_test` to `tls_test`.

### 6.3.4. Client does not provide certificate

OpenSSL client:

```
"$ openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys"
```

---

```

$ ./scripts/tls_test --listen localhost --port 8443 --key caKey.der --cert cert.der
↪ --cacert cert.der
127.0.0.1[53256] connected
using key of type RSA
negotiated TLS 1.3 using suite TLS_AES_256_GCM_SHA384
using key exchange CURVE25519
sending TLS server certificate 'C=CH, O=strongSwan, CN=localhost'
no certificate sent by peer
sending fatal TLS alert 'decode error'
127.0.0.1[53256] disconnected
^C

```

---

Listing 39: Client does not provide certificate and server terminates connection.

Protocol	Length	Info
TLSv1.3	349	Client Hello
TLSv1.3	1381	Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate Request, Certificate, Certificate Verify, Finished
TLSv1.3	176	Change Cipher Spec, Certificate, Finished
TLSv1.3	90	Alert (Level: Fatal, Description: Decode Error)

Figure 6.15.: Client does not provide certificate.

## 7. Results

All the mandatory goals in this bachelor thesis were achieved. The TLS library `libtls` within `strongSwan` now supports the TLS 1.3 protocol both on the client and the server side. It is also possible to authenticate a connecting client by using X.509 certificates in a mutual authentication handshake scenario. The code was heavily tested in existing and new unit tests, with `OpenSSL` as TLS reference implementation and with external web services such as Google.

The optional goal, PSK-based TLS 1.3 session resumption, was not completely implemented within this theses. However, the foundation has been laid: It is possible to generate all the necessary cryptographic secrets and keys with the HKDF implementation. The added unit test cases for 1-RTT as well as 0-RTT handshake scenarios prove the correctness of the HKDF implementation for PSK-based requirements.

Although `FreeRADIUS` version 3.0.20 from November 14th 2019 states that it does not currently support TLS 1.3<sup>1</sup>, integration and regression tests<sup>2</sup> with the latest `FreeRADIUS` version behave correctly. Also, the master session key (MSK), derived by the HKDF implementation, is computed correctly and compatible with `FreeRADIUS`. Nevertheless, additions in the `strongSwan` EAP layer for the use of TLS 1.3 in accordance with the EAP-TLS 1.3 draft<sup>34</sup> are required.

Section 7.1 provides more specific information about the tasks accomplished during this bachelor thesis.

### 7.1. Achievements

**Implementation of the TLS 1.3 server-side protocol** The TLS 1.3 server-side protocol stack was successfully implemented and tested. In addition, support for `HelloRetryRequest` (HRR) messages was implemented to allow two peers to negotiate a common DH group. Also, passive support for rekeying was implemented by supporting `KeyUpdate` messages. If another peer likes to set new keys, `libtls` can handle this on inbound and outbound channels. Several new unit tests verify correct functionality of all TLS versions' interconnection between `libtls` client and server.

**Implementation of TLS 1.3 client authentication based on X.509 client certificates** The `libtls` client implementation was enhanced to support X.509 based client authentication to achieve TLS 1.3 mutual authentication. In addition to existing support for RSA

---

<sup>1</sup>because of draft status of the standards[11],[5] [https://freeradius.org/release\\_notes/](https://freeradius.org/release_notes/) visited 2020-12-16

<sup>2</sup><https://wiki.strongswan.org/projects/strongswan/wiki/TestingEnvironment> visited 2020-12-16

<sup>3</sup>11.

<sup>4</sup>5.

and ECDSA keys, support for Ed25519 and Ed448 keys were implemented, both for client and server authentication. All these features were supplemented by corresponding unit tests to prove correct behaviour and functionality.

**Interoperability testing of TLS 1.3 libtls stack with 3rd party implementations** Client and server implementation of libtls were heavily tested with OpenSSL. Not only correct handshakes were tested but also incorrect behaviour as described in the manual testing chapter 6.

**PSK-based TLS 1.3 session resumption (0-RTT)** Session resumption in TLS 1.2 saves key computing time and speeds connection establishment up. This goal was inspired by these thoughts. However, RFC 8446's PSK-based resumption mainly focuses on the 0-RTT early data feature which is not a requirement for strongSwan. It is possible though to use PSK-based resumption without the burden of computing new (EC)DHE keys and establish a connection without a `key_share` extension and therefore without forward secrecy, but RFC 8446 does recommend to always send a `key_share` extension and therefore compute (EC)DHE keys as described in section 4.3.1. During this thesis, it became clear that PSK-based resumption is rather understood as a long term authentication feature that omits the need of sending potentially many bytes long certificates over the wire. Further discussion and reevaluation is required to determine the need for this feature in the strongSwan context. However, the cryptographic foundation in the HKDF has been laid to generate all the needed key material.

Also many smaller additions, improvements and fixes were achieved:

- Rework cipher suite preference order
- Support multiple client exchange key shares
- Support Edwards-curve Digital Signature Algorithm (EdDSA) keys in ECDSA cipher suites
- config file support for DH group
- Refactor writing of key share extensions
- Add signature scheme constants
- Check if peer sent signature algorithms extension
- Generalise trusted public and private key search
- Terminate connection if peer certificate is required but not sent
- Implement resumption key and binder PSK generation in HKDF
- new TLS 1.2 features:
  - support three additional cipher suites<sup>5</sup>
  - support three additional extensions<sup>6</sup>
  - Make CertificateRequest conditional in old TLS versions

---

<sup>5</sup>TLS\_ECDHE\_RSA\_WITH\_CHACHA20\_POLY1305\_SHA256,  
TLS\_ECDHE\_ECDSA\_WITH\_CHACHA20\_POLY1305\_SHA256 &  
TLS\_DHE\_RSA\_WITH\_CHACHA20\_POLY1305\_SHA256

<sup>6</sup>Encrypt then MAC, Extended master secret & Session ticket

- Fixes:
  - Fix invalid signature algorithm list building
  - Fix invalid signature algorithm and supported groups parsing
  - Fix typo in HKDF label
  - Fix missing client finished handshake bytes

## 7.2. Further work

A lot was achieved during the past 16 weeks of this bachelor thesis. However, some of the optional tasks were not accomplished due to the lack of time. The following list provides an overview of open tasks:

- *PSK-based resumption*<sup>7</sup> The implementation achieved during this work covers the cryptographic part. The HKDF is able to generate all the needed secrets and was tested against test data provided by RFC 8448.<sup>8</sup> The message communication protocol in `tls_peer.c` and `textttls_server.c` however has not been implemented yet.
- *Downgrade protection*<sup>9</sup> TLS 1.3 defines a downgrade attack protection embedded in the server's `ServerHello` message random value as described in.<sup>10</sup>
- *Configure ciphers by configuration file*<sup>11</sup> Allows the configuration of cipher suites available to strongSwan by configuration file.
- *Configure signature algorithms by configuration file*<sup>12</sup> Allows the configuration of signature algorithms available to strongSwan by configuration file.
- *Apply order of DH groups configured in configuration file*<sup>13</sup> The available DH groups can be configured by the configuration file but the ordering of the entries in the configuration file is not considered.
- *Enhance Diffie-Hellman group to TLS group mapping with finite field groups*<sup>14</sup> Since other TLS 1.3 implementations such as Firefox offer also finite field groups, it would make sense to also provide these groups in our implementation.
- *Rename "curve" to be more accurate with the changes in TLS 1.3 "supported" group naming convention*<sup>15</sup> This is a more cosmetic change to improve readability and consistency with RFC 8446.
- *Implement missing CHACHA20 POLY1305 TLS 1.2 cipher suites*<sup>16</sup> The three CHACHA20 POLY1305 ciphers could be implemented as described in RFC 7905.<sup>17</sup>

---

<sup>7</sup>Implement PSK resumption #4

<sup>8</sup>24.

<sup>9</sup>Implement downgrade protection #1

<sup>10</sup>13, pp. 32–33.

<sup>11</sup>Configure ciphers via configuration file for TLS 1.3 #15

<sup>12</sup>Configure signature algorithms via config file #27

<sup>13</sup>Apply config order of DH groups #23

<sup>14</sup>Enhance diffie hellman group to tls group mapping with finit field groups #20

<sup>15</sup>Rename curve to be more accurate with the changes in TLS 1.3 supported group naming convention #22

<sup>16</sup>Implement missing CHACHA20 POLY1305 TLS 1.2 cipher suites #13

<sup>17</sup>10.

As discussed in section 2.3.1, the main use for the TLS stack in strongSwan is for the EAP authentication scenarios. Since TLS 1.3 works differently than predecessor versions, EAP needs adjustments to work with the new TLS version. However, currently no finished RFC exists that specifies EAP with TLS 1.3. The two relevant drafts are “Using EAP-TLS with TLS 1.3”<sup>18</sup> and “TLS-based EAP types and TLS 1.3”.<sup>19</sup>

## 7.3. Outlook to TLS 1.4?

TLS 1.2 was released in 2008<sup>20</sup> and it took ten years to release the RFC describing TLS 1.3 in 2018,<sup>21</sup> so TLS in version 1.4 does not lurk around the corner yet. However, the TLS-WG<sup>22</sup> which standardises the development in the TLS ecosystem is continuously working on TLS 1.3 related updates and improvements. The following list<sup>23</sup> of working titles gives the reader an impression in what direction the upcoming improvements will lead.

**TLS Ticket Requests** “TLS session tickets enable stateless connection resumption for clients without server-side, per-client, state. Servers vend an arbitrary number of session tickets to clients, at their discretion, upon connection establishment. Clients store and use tickets when resuming future connections.”<sup>24</sup>

**A Flags Extension for TLS 1.3** “A number of extensions are proposed in the TLS working group that carry no interesting information except the 1-bit indication that a certain optional feature is supported. Such extensions take 4 octets each. This document defines a flags extension that can provide such indications at an average marginal cost of 1 bit each.”<sup>25</sup>

**Importing External PSKs for TLS** “This document describes an interface for importing external Pre-Shared Keys (PSKs) into TLS 1.3.”<sup>26</sup>

**Encrypted Server Name Indication for TLS 1.3** “This document describes a mechanism in Transport Layer Security (TLS) for encrypting a ClientHello message under a server public key.”<sup>27</sup>

**TLS Certificate Compression** “In TLS handshakes, certificate chains often take up the majority of the bytes transmitted. This document describes how certificate chains can be compressed to reduce the amount of data transmitted and avoid some round trips.”<sup>28</sup>

**Exported Authenticators in TLS** “This document describes a mechanism in Transport Layer Security (TLS) for peers to provide a proof of ownership of an identity, such as an X.509

---

<sup>18</sup>11.

<sup>19</sup>5.

<sup>20</sup>14.

<sup>21</sup>13.

<sup>22</sup><https://datatracker.ietf.org/wg/tls/about/> visited 2020-12-15

<sup>23</sup><https://datatracker.ietf.org/wg/tls/charter/> visited 2020-11-28

<sup>24</sup><https://datatracker.ietf.org/doc/draft-ietf-tls-ticketrequests/> visited 2020-11-28

<sup>25</sup><https://datatracker.ietf.org/doc/draft-ietf-tls-tlsflags/> visited 2020-11-28

<sup>26</sup><https://datatracker.ietf.org/doc/draft-ietf-tls-external-psk-importer/> visited 2020-11-28

<sup>27</sup><https://datatracker.ietf.org/doc/draft-ietf-tls-esni/> visited 2020-11-28

<sup>28</sup><https://datatracker.ietf.org/doc/rfc8879/> visited 2020-12-15

certificate. This proof can be exported by one peer, transmitted out-of-band to the other peer, and verified by the receiving peer.”<sup>29</sup>

#### **Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility**

“This document describes GREASE (Generate Random Extensions And Sustain Extensibility), a mechanism to prevent extensibility failures in the TLS ecosystem. It reserves a set of TLS protocol values that may be advertised to ensure peers correctly handle unknown values.”<sup>30</sup>

**Delegated Credentials for TLS** “The organizational separation between the operator of a TLS endpoint and the certification authority can create limitations. For example, the lifetime of certificates, how they may be used, and the algorithms they support are ultimately determined by the certification authority. This document describes a mechanism by which operators may delegate their own credentials for use in TLS, without breaking compatibility with peers that do not support this specification.”<sup>31</sup>

**Importing External PSKs for TLS** “This document describes an interface for importing external Pre-Shared Keys (PSKs) into TLS 1.3.”<sup>32</sup>

**Deprecating TLSv1.0 and TLSv1.1** “This document, if approved, formally deprecates Transport Layer Security (TLS) versions 1.0 (RFC 2246) and 1.1 (RFC 4346).”<sup>33</sup>

In addition, the new transport protocol QUIC, currently standardised by the QUIC-WG<sup>34</sup>, is based on TLS 1.3.

In summary, TLS 1.3 is standardised in RFC 8446 and is used in the wild. However, its ecosystem is constantly developed and improved as the list of RFC’s and drafts above shows. TLS 1.3 is also a very important foundation for other protocols on the application layer, such as HTTP or DNS, but also on the transport layer as in QUIC. TLS 1.3 is a solid ground to build the future of secure transport communication protocols.

---

<sup>29</sup><https://datatracker.ietf.org/doc/draft-ietf-tls-exported-authenticator/> visited 2020-12-15

<sup>30</sup><https://datatracker.ietf.org/doc/rfc8701/> visited 2020-12-15

<sup>31</sup><https://datatracker.ietf.org/doc/draft-ietf-tls-subcerts/> visited 2020-12-15

<sup>32</sup><https://datatracker.ietf.org/doc/draft-ietf-tls-external-psk-importer/> visited 2020-12-15

<sup>33</sup><https://datatracker.ietf.org/doc/draft-ietf-tls-oldversions-deprecate/> visited 2020-12-15

<sup>34</sup><https://datatracker.ietf.org/wg/quic/about/> visited 2020-12-15

## A. List of Abbreviations

<b>AEAD</b>	Authenticated Encryption with Associated Data
<b>AES</b>	Advanced Encryption Standard
<b>CA</b>	Certificate Authority
<b>DHE</b>	Ephemeral Diffie-Hellman
<b>DSA</b>	Digital Signature Algorithm
<b>EAP</b>	Extensible Authentication Protocol
<b>EAP-PEAP</b>	EAP with Protected Extensible Authentication Protocol
<b>EAP-TLS</b>	EAP with Transport Layer Security
<b>EAP-TTLS</b>	EAP with Tunneled Transport Layer Security
<b>ECDHE</b>	Elliptic Curve DHE
<b>ECDSA</b>	Elliptic Curve DSA
<b>EdDSA</b>	Edwards-curve DSA
<b>GCM</b>	Galois/Counter Mode
<b>HKDF</b>	HMAC-based Extract-and-Expand Key Derivation Function (RFC 5869)
<b>HMAC</b>	Keyed-Hash Message Authentication Code
<b>HRR</b>	Hello Retry Request is a TLS 1.3 message type
<b>IETF</b>	Internet Engineering Task Force
<b>IKE</b>	Internet Key Exchange
<b>IKM</b>	Input Key Material
<b>IP</b>	Internet Protocol
<b>IPsec</b>	IP Security
<b>IV</b>	Initialisation Vector
<b>MAC</b>	Message Authentication Code
<b>MSK</b>	Master Session Key
<b>Nonce</b>	Number used only once
<b>OKM</b>	Output Key Material
<b>OSI</b>	Open Systems Interconnection Model
<b>PKI</b>	Public Key Infrastructure



**PRF** Pseudo-Random Function  
**PSK** Pre-Shared Key  
**PT-TLS** Posture Transport Protocol over TLS  
**RFC** Request For Comment  
**RSA** Rivest-Shamir-Adleman  
**RSA-PSS** RSA Probabilistic Signature Scheme  
**RTT** Round-Trip Time  
**SHA** Secure Hash Algorithm  
**TCP** Transmission Control Protocol  
**TLS** Transport Layer Security

## B. Bibliography

- [1] Aumasson, Jean-Philippe. *Serious Cryptography*. No Starch Press, 2018 (cit. on pp. 12, 13, 15).
- [2] Beurdouche, Benjamin and Bhargavan, Karthikeyan et al. "A messy state of the union: taming the composite state machines of TLS". In: *Commun. ACM* 60.2 (2017), pp. 99–107. DOI: 10.1145/3023357. URL: <https://doi.org/10.1145/3023357> (cit. on p. 14).
- [3] Boneh, Dan. *Cryptography: Authenticated Encryption*. 2015. URL: <https://www.youtube.com/watch?v=40m3gcdGDu0> (visited on 03/11/2020) (cit. on p. 40).
- [4] Boneh, Dan. *Key Derivation*. 2020. URL: <https://www.coursera.org/lecture/crypto/key-derivation-A1ETP> (visited on 03/11/2020) (cit. on p. 26).
- [5] DeKok, Alan. *TLS-based EAP types and TLS 1.3*. Internet-Draft. IETF Tools, July 2020, pp. 1–12. URL: <https://tools.ietf.org/html/draft-ietf-emu-tls-eap-types-01> (cit. on pp. 75, 78).
- [6] Fall, Kevin R. and Stevens, Richard W. *TCP/IP illustrated, Volume 1. The Protocols*. 2nd ed. Addison-Wesley, 2012 (cit. on pp. 12, 16).
- [7] Holz, Ralph et al. *The Era of TLS 1.3: Measuring Deployment and Use with Active and Passive Methods*. 2019. arXiv: 1907.12762 [cs.CR] (cit. on p. 15).
- [8] Knecht, Pascal and Sieber, Méline. *TLS 1.3 for strongSwan: A Client-Side Prototype*. Study Project. Hochschule für Technik Rapperswil, May 2020, pp. 1–64 (cit. on pp. 12, 17, 31, 32, 38).
- [9] Krawczyk, Hugo. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. IETF Tools, May 2010, pp. 1–14. URL: <https://tools.ietf.org/html/rfc5869> (cit. on pp. 15, 26).
- [10] Langley, Adam et al. *ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)*. RFC 7905. IETF Tools, June 2016, pp. 1–8. URL: <https://tools.ietf.org/html/rfc7905> (cit. on p. 77).

- [11] Mattsson, John Preuss and Sethi, Mohit. *Using EAP-TLS with TLS 1.3*. Internet-Draft. IETF Tools, Nov. 2020, pp. 1–30. URL: <https://tools.ietf.org/html/draft-ietf-emu-eap-tls13-13> (cit. on pp. 75, 78).
- [12] Moriaty, K. and Farrell, S. *Deprecating TLSv1.0 and TLSv1.1*. Draft 5216. IETF Tools, 2020, pp. 1–23. URL: <https://tools.ietf.org/html/draft-ietf-tls-oldversions-deprecate-09> (cit. on p. 32).
- [13] Rescorla, Eric. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. IETF Tools, Aug. 2018, pp. 1–160. URL: <https://tools.ietf.org/html/rfc8446> (cit. on pp. 12, 14, 15, 23, 26–29, 36, 37, 41, 77, 78).
- [14] Rescorla, Eric and Dierks, Tim. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. IETF Tools, Aug. 2008, pp. 1–104. URL: <https://tools.ietf.org/html/rfc5246> (cit. on pp. 26, 40, 78).
- [15] Ristić, Ivan. *Bulletproof SSL and TLS. Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*. Feisty Duck, 2017 (cit. on p. 13).
- [16] Sangster, Paul. *A Posture Transport Protocol over TLS (PT-TLS)*. RFC 6876. IETF Tools, Oct. 2015, pp. 1–44. URL: <https://tools.ietf.org/html/rfc6876> (cit. on p. 17).
- [17] Schmidt, Jürgen. “Weniger ist mehr. Was die anstehende Version TLS 1.3 bringt”. In: *c’t* 4 (2017), pp. 172–174 (cit. on p. 12).
- [18] Simon, Dan et al. *The EAP-TLS Authentication Protocol*. RFC 5216. IETF Tools, Mar. 2008, pp. 1–34. URL: <https://tools.ietf.org/html/rfc5216> (cit. on p. 16).
- [19] Stallings, William. *Network Security Essentials. Applications and Standards*. Pearson Education Limited, 2017 (cit. on pp. 12, 16, 17).
- [20] Steffen, Andreas. *Advanced Features of Linux strongSwan*. Tech. rep. 2005. URL: <https://strongswan.org/docs/LinuxTag2005-strongSwan.pdf> (visited on 03/11/2020) (cit. on p. 12).
- [21] strongSwan. *About*. 2018. URL: <https://wiki.strongswan.org/about.html> (visited on 03/11/2020) (cit. on p. 16).
- [22] strongSwan. *EAP-TLS*. 2015. URL: <https://wiki.strongswan.org/projects/strongswan/wiki/EapTls> (visited on 03/11/2020) (cit. on p. 16).
- [23] strongSwan. *IKE keying daemon charon*. 2017. URL: <https://wiki.strongswan.org/projects/strongswan/wiki/Charon> (visited on 03/11/2020) (cit. on p. 16).
- [24] Thomson, Martin. *Example Handshake Traces for TLS 1.3*. RFC 8448. IETF Tools, Jan. 2019, pp. 1–68. URL: <https://tools.ietf.org/html/rfc8448> (cit. on pp. 57, 77).
- [25] Wikipedia. *Extensible Authentication Protocol*. URL: [https://en.wikipedia.org/wiki/Extensible\\_Authentication\\_Protocol](https://en.wikipedia.org/wiki/Extensible_Authentication_Protocol) (visited on 03/11/2020) (cit. on p. 16).

- [26] Wikipedia. *Transport Layer Security*. URL: [https://de.wikipedia.org/wiki/Transport\\_Layer\\_Security#TLS\\_Change\\_Cipher\\_Spec\\_Protocol](https://de.wikipedia.org/wiki/Transport_Layer_Security#TLS_Change_Cipher_Spec_Protocol) (visited on 27/11/2020) (cit. on p. 12).

## C. Compile Instructions

In this bachelor thesis we worked with the code base of the official strongSwan GitHub repository. Initially we created a fork in my GitHub name space and followed the official compile instructions of the strongSwan wiki.

### C.1. Compile and Unit Tests

On our Ubuntu 20.04.1 LTS machine the following additional packages had to be installed in order to compile the project:

- autoconf
- autogen
- build-essential
- libtool
- gperf
- libssl-dev
- lcov

---

```
1 $ sudo apt install autoconf autogen build-essential libtool gperf libssl-dev
```

---

Listing 40: Installation of required packages.

The following steps clone the project source code, build it and run all unit tests.

---

```
$ git clone https://github.com/ryru/strongswan.git
$ mkdir build
$ cd strongswan/
$ git checkout tls13
$ ./autogen.sh
$ cd ../build/
$ ../strongswan/configure --prefix=/usr --sysconfdir=/etc --disable-defaults \
--enable-silent-rules --enable-leak-detective --enable-scripts --enable-openssl \
--enable-wolfssl --enable-botan --enable-x509 --enable-pkcs1 --enable-pkcs8 \
--enable-pem --enable-eap-tls
$ make -j
$ sudo make install
$ make check
```

---

Listing 41: Building steps to compile the project.

The above procedure is slightly different than described in the strongSwan wiki. We built the project in a separate build directory outside of the project source code to keep generated object and binary files outside of the source tree.

The used configure flags are an absolute minimum configuration to develop and run all the relevant components in this thesis. A short description based on `./configure --help` follows:

- `--prefix=/usr` Install architecture-independent files in `/usr`. This is relevant in combination with `sudo make install`.
- `--sysconfdir=/etc` Read-only single-machine data shall be stored under `/usr/etc`. This is relevant in combination with `sudo make install`.
- `--disable-defaults` Disable all default plug-ins. This allows faster initial compile time and a smaller binary footprint.
- `--enable-silent-rules` Less verbose build output which is a personal preference.
- `--enable-leak-detective` Enable malloc hooks to find memory leaks.
- `--enable-scripts` Enable additional utilities. This is relevant for the `tls_test` application.
- `--enable-openssl` Enables the OpenSSL crypto plug-in which is used for its cryptographic primitives.
- `--enable-wolfssl` Enable WolfSSL as crypto backend. This flag is not required for a minimal setup and requires `libwolfssl-dev` installed on your system.
- `--enable-botan` Enable Botan as crypto backend. This flag is not required for a minimal setup and requires `libbotan-2-dev` installed on your system.
- `--enable-x509` Enable support for X.509 encoded certificates. This flag is not required for a minimal setup and only is required when OpenSSL is not used.
- `--enable-pkcs1` Enable support for PKCS1 encoding. This flag is not required for a minimal setup and only is required when OpenSSL is not used.

- `--enable-pem` Enable PEM decoding plug-in, which is used to handle certificates in the PEM format.
- `--enable-eap-tls` Enable EAP TLS authentication module, this is used to build the TLS library (libtls) itself.
- `--enable-coverage` Used to generate local coverage reports. Requires `lcov`.

To use all cores on my Lenovo ThinkPad T470 I use the `-j` flag while building with `make`. This reduces compilation from one minute in single job mode to about 23 seconds in multi jobs mode.

For a minimal setup with OpenSSL as cryptographic back-end, it is not required to install the project via `sudo make install`. This is only required if other back-ends such as “Botan” or “WolfSSL” are used. Because both do not ship with X.509 support, strongSwan’s own implementation `x509` is needed which has to be installed before it can be referenced.

To easily switch between cryptographic back-ends one can use the environment variable `PLUGINS` with `tls_test`:

- `export PLUGINS="pem openssl"` this is the default when using a minimal setup.
- `export PLUGINS="pem x509 pkcs1 wolfssl"` to use WolfSSL
- `export PLUGINS="pem x509 pkcs1 botan"` to use Botan

To test more specifically `make check` can be parametrised as described in the official Wiki. We used combinations of the following flags:

- `TESTS_RUNNERS=libtls` only run the “libtls” test runner with all its test suites.
- `TESTS_SUITES='HKDF TLS 1.3'` only run the test suite “HKDF TLS 1.3” which is within the libtls test runner.
- `TESTS_CASES='TLS 1.2/anon'` only run the test case “TLS 1.2/anon” which is part of the “socket” test suite within the “libtls” test runner.
- `LEAK_DETECTIVE_DISABLE=1` deactivates the memory leak detection mechanism for faster test execution time and fewer output to the console.
- `TESTS_VERBOSE=-1` specify how detailed the output to “STDOUT” is. This is a very important debugging flag with these possible values:
  - `-1` absolutely silent no output to “STDOUT”. This also represents the default value when this flag is not specified.
  - `0` very basic auditing logs.
  - `1` generic control flow with errors.
  - `2` more detailed debugging control flow.
  - `3` including raw data dumps in hex.
  - `4` also include sensitive material in dumps, e.g. keys.

### C.1.1. Code Coverage

To render code coverage results loyally, one follows these steps in the build directory:

---

```
../workspace/configure --prefix=/usr --sysconfdir=/etc --enable-silent-rules  
↪ --enable-scripts --enable-pem --enable-eap-tls --enable-x509 --enable-pkcs1  
↪ --enable-openssl --enable-coverage  
make clean  
make coverage
```

---

Listing 42: Generate code coverage report.

The output is generated in the new directories `coverage/html`.



## D. Source Code

### D.1. Code-Repository

The complete source code of this implementation can be found online here:

<https://github.com/ryru/strongswan/>

Interesting branches are:

- `tls13`: the final code

## E. OpenSSL Commands

---

```
$ openssl version
OpenSSL 1.1.1f 31 Mar 2020
```

---

Listing 43: OpenSSL version used in this chapter.

### E.1. Handshake TLS 1.3 client to TLS 1.3 server

---

```
$ openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys -keyform
↪ der -key caKey.der -certform der -cert cert.der
```

---

Listing 44: TLS 1.3 server.

---

```
$ openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
↪ --keyform der --key caKey.der --certform der --cert cert.der
```

---

Listing 45: TLS 1.3 client.

### E.2. Handshake TLS 1.2 client to TLS 1.3 server

---

```
$ openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys -keyform
↪ der -key caKey.der -certform der -cert cert.der
```

---

Listing 46: TLS 1.3 server.

---

```
$ openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys
↪ --keyform der --key caKey.der --certform der --cert cert.der -tls1_2
```

---

Listing 47: TLS 1.2 client.

## E.3. Handshake TLS 1.3 client to TLS 1.2 server

---

```
$ openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys -keyform  
↪ der -key caKey.der -certform der -cert cert.der -tls1_2
```

---

Listing 48: TLS 1.2 server.

---

```
$ openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys  
↪ --keyform der --key caKey.der --certform der --cert cert.der
```

---

Listing 49: TLS 1.3 client.

## E.4. KeyUpdate messages

The commands are the same as in section E.1. Once the connection is established the two interactive commands `k` and `K` are used:

- `k` Send a key update message to the client.
- `K` Send a key update message to the client and request one back.

## E.5. HelloRetryRequest

---

```
$ openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys -keyform  
↪ der -key caKey.der -certform der -cert cert.der --curves secp521r1
```

---

Listing 50: TLS 1.3 server expects *secp521r1* key share DH group.

---

```
$ openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys  
↪ --keyform der --key caKey.der --certform der --cert cert.der
```

---

Listing 51: TLS 1.3 client by default sends *x25519* key share DH group.

## E.6. Mutual authentication

---

```
$ openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys  
↪ --keyform der --key caKey.der --certform der --cert cert.der --verify 3
```

---

Listing 52: TLS 1.3 server expects client to authenticate itself.

---

```
$ openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys  
↪ --keyform der --key caKey.der --certform der --cert cert.der
```

---

Listing 53: TLS 1.3 client authenticates itself with the key material.

## E.7. PSK session resumption

---

```
openssl s_server -accept localhost:8443 -debug -keylogfile ~/tls13-secrets.keys -keyform  
↪ der -key caKey.der -certform der -cert cert.der
```

---

Listing 54: Starts an OpenSSL server instance.

The file `client.session` needs to exist.

---

```
openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys --keyform  
↪ der --key caKey.der --certform der --cert cert.der -sess_out client.session
```

---

Listing 55: Starts an OpenSSL client instance and writes session ticket into `client.session` file.

---

```
openssl s_client -connect localhost:8443 -debug -keylogfile ~/tls13-secrets.keys --keyform  
↪ der --key caKey.der --certform der --cert cert.der -sess_out client.session -sess_in  
↪ client.session -psk_session client.session
```

---

Listing 56: Starts an OpenSSL client instance and reads session ticket from `client.session` file.

---

```

-----BEGIN SSL SESSION PARAMETERS-----
MIIeAQIBAQICAwQEAhMCBCCGBJgwqsovf3gvDBtnBliLkZDq0BBf+ur634b3I8mx
3QQwr/y+PLB3u0dnWtm+AgL4fGRyjZrrcdsizr9zGJWuZJlMp+rIkEp1ezRhZ8zkQ
KEmMoQYCBF+1JKeiBAICHCCjggMKMIIDBjCCAe6gAwIBAgIA4zomyXxsXswDQYJ
KoZIHvcNAQELBQAwnjELMAkGA1UEBhMCQ0gxZzARBGNVBAoTCnN0cm9uZ1N3YW4x
EjAQBGNVBAMTCWxvY2FsaG9zdDAeFw0yMDA5MTEwODQxMjNaFw0yMzA5MTEwODQx
MjNaMDYxZzAJBgNVBAYTAkNIMRMwEQYDVQQKEwpzdHJvbmdd2FuMRIwEAYDVQQD
Ewlsb2NhbGhvc3QwggeiMAOGCSqGSIB3DQEBQUAA4IBDwAwggEKAoIBAQCxgIvf
Fb+IhtpKpbI+tfAbAyazNbZP8yH3jjzawxkCK40d89HtboJRLsZvztVmH9hy8j3A
vorDTORj2PPJT9M3iLRHBj216C4MsxZ4Hq90mVLQpvlbrKDxQyyPbk/Q/MdtpeSC
GD/7w3GBtSyoQcuReEJ32dQ4a6iMJ138Ai01bIlWBFfvDYzDM8TM0fL45gdN9RHL
YiEovj9BZ3tgb+85yWdYWi0tyvsbMXj/23cIV9kS+GnxhUw/PnMeDQPGheo7fnoL
5P1WMQh1vfoQH91xEo9XBQoKAf96hoG0A3fjD3P7Ab7xLvSqyNxjPRLmMmXp1IYU
0cILbSCcsqKn3dUjAgMBAAQjGDAWMBQGA1UdEQQNMAuCCWxvY2FsaG9zdDANBgkq
hkiG9w0BAQsFAAOCQAQEAiFqVRHMKdRwTz07ilxBfkmHqniEifGMVnVLCj/4uoF1o
ns6dgi7wjwiDNwrgUCXE3wB3SHSDAmuUzfrq/g+WWOMkBFra+cb2JBUfs0E6CiF4
giDC8JcDdqEFVaKgbYVwGNTntShOQ/cbHFXNrCf11dQLInTEmdLeuvWml517vDXh
oUuw25xIcM69NBvIe5IiSnTxF21dNEhaxyybnOnXhIUeL+kYrecoCjm1wzh0BWzW
pU6tQdCBaPYv709YuQww11U7rT43LBCz3Aeb8WmMm6vwJSWovrD/nLeLicBsEmV0
rNomxIkR1FnNFQaBjk0F4gcJON0g3RfglPNnGs/nIaQCBAC1AwIBEqkEAgIcIKqB
0wSBOMv1R8thifw0Vn8BGFJ/wM9CqNgcWgck3C6yoVs0lVMG8j5+01tCs3W+PAZe
o8pNmLx2m2TiN20CxpFAvTjdJ4m0R1I//oo1/Ugbv3TWvGVNJ81OrzqhOoupJKET
Y2GYfuuEEHyg7OPLw110WLg0YJnDv+0/W1YnNig9RxokfECp3M3Qj+pvhJH290Qq
4pLw0faJy1mo5kPXJj3GMkyebsqQ++wksSzSkBJeARYtmuSH+eMD6Qt4+rXzoWA
DIvuLfWqpnmASFm9+/rNHY7wXmGuBwIFALwg4Es=
-----END SSL SESSION PARAMETERS-----

```

---

Listing 57: Content of the session ticket in file client.session.

## F. Key material

This chapter describes how key material used mainly for testing purpose was generated. Additionally, the concrete private key and X.509 certificate used during this thesis is included due to traceability and reproduction purpose.

---

```
pki --gen --type rsa
pki --self --in rsa.key --dn "C=CH, O=strongSwan, CN=localhost"
```

---

Listing 58: Key generation using strongSwan's pki tool.

---

```
$ openssl rsa -inform der -in caKey.der
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAsYCL3xW/iIbaSqWyPrXwGwMmszW2T/Mh94482sMZAiuNHfPR
7W6CUS7Gb87VZh/YcvI9wL6Kw09EY9jzyU/TN4i0RwY9teguDLMWeB6vdJlSOKb5
W6yg8UMsj25P0PzHbaXkghg/+8NxbUsqEHLkXhCd9nU0GuojCdd/AitNWyJVgRX
7w2MwzPEzNHY+OYHTfURy2IhKL4/QWd7YG/v0cInWFOjrcr7GzF4/9t3CFfZEvhP
8YVMPz5zHg0DxoXq0356C+T9VjEIdb36EB/dcRKPvWUKCgH/eoaBtAN34w9z+wG+
8S70qsjcYz0S5jJl6dSGFNHCC20gnLKip93VIwIDAQABAoIBAQCBCOu1Xs//1dh2
j4HGKMjF0kM14AQKlrrI1bQTa+SQOUy6y3Z+XemKTrCeswNa3CejWAogRePmG4eh
9iMy4z6ujkhPoW/W/1QKk13wMI5n+tXDLL71L8dH3AdWtWB8kmX/hataD3rj8K5h
Fm3CWyI70+tf6SpnYYwDZtLh0ks/G9Ij2prdZzM96yMpCKVtj0dFAQkeb/0zKC9F
6J0lpp4FbeD70okDifu0a4iZ/4H2Gq6zjBkEGArwuEVq8hPHxjhELy60VzBZDdMs
Onjc0Ka4LmEL3VIkkC08mHj0w7GwMLwc+rL+KYsiFd6vTbEEHgDPPHG+XoDLp0hX
3Jhwsu7hAoGBAOWsMDNezfCarwrEkk3aP8T/tIdN0/qs851NhLcvYYsEu8/Ajzjd
c4mi6pKRal00z49/bxtQdQfyI6See0Z6/R2t6ZVKSwW0cDFMBnYLEJaV8eznAV9k
Uc494P3f1M7khKe1E95mlwJlu3EcXaljIq8o0L3HA0wcKaamA/ua1rx5AoGBAMbn
aiohWghv7dXl1p0k/8EN3cQr5R2U07EV/HxkEehL0L3Pa7epJ0AOSWYNaz58XIhj
5v0wQ70PIc1UxtXk2/fsz+kL+xKMn5PuOXBCL9AnIkLt481e7f71FJcNZcc0S0WV
XYamfUml8ZzmPDsQsvE+A9o27Hhh+Yc0zcYR6797AoGAHLBNdQAIkjaofr3EK/Lc
FjkdM+YHXwcNXEz/VyVJSxLbIar5gWXD0BXzOrSxX/sfT4YiYAYJAVsXm1Koyw6b
mrRakIkCzfz8gfCDRGmZrUEvryBsmULdcRkxMIW1GyDFQH4NmNGLfH4RI+9XLVFe
kDp8COGYqhXNtXGrQDAR77kCgYBPsIWAPaAUrodLeI6TN6fWRe/8OFIFFkhC6GBT
BP+FvyEN/MMYpq7jIMTlx9kLYGbS9s3JUNA9le7iJKLWJJwW67aRR60N2cCefKS9
rPEAvbSrE+yuOwtH74m9ABhfkYA/1sWN55csxxGjcpivvhLfaFggIW9/zNSU48TG
rNV/0QKBgBvgExpW2w0VoId7JpQabIq1tUfKDwDEZvd0EJQAv4ZVKS2BQ4+3nms2
Yq4N7h/c/8510W0DvrY/j0TgbQ9nTCNNVPMoOoh78HJECUxvHqWcjrR08A+Vc9Qo
j98yPOUCUXUYU3a6j0GgzmSWWTMa89kXif1ftfhzoP/rgixuUA2v
-----END RSA PRIVATE KEY-----
```

---

Listing 59: RSA private key used during this bachelor thesis.

---

```

$ openssl x509 -inform der -in cert.der
-----BEGIN CERTIFICATE-----
MIIDBJCCAE6gAwIBAgIIA4zomyXxsXswDQYJKoZIhvcNAQELBQAwnJELMAkGA1UE
BhMCQ0gxZzARBgNVBAoTCnN0cm9uZ1N3YW4xEjAQBgNVBAMTCWxvY2FsaG9zdDAe
Fw0yMDA5MTEwODQxMjNaFw0yMzA5MTEwODQxMjNaMDYxCzAJBgNVBAYTAkNIMRMw
EQYDVQQKEwpzdHJvbmddTd2FuMRIwEAYDVQQDEwlsb2NhbGhvc3QwggEiMA0GCSqG
SIb3DQEBAQUAA4IBDwAwggEKAoIBAQCxgIvfFb+IhtpKpBI+tfAbAyazNbZP8yH3
jjzawxkCK40d89HtboJRLsZvztVmH9hy8j3AvorDTORj2PPJT9M3iLRHBj216C4M
sxZ4Hq90mVLQpvlbrKDxQyyPbk/Q/MdtpeSCGD/7w3GBtSyoQcuReEJ32dQ4a6iM
J138Ai01bIlWBffvDYzDM8TM0fL45gdN9RHLyIEovj9BZ3tgb+85yWdYWi0tyvsb
MXj/23cIV9kS+GnxhUw/PnMeDQPGheo7fnoL5P1WMQh1vfoQH91xEo9XBQoKaf96
hoGOA3fjD3P7Ab7xLvSqyNxjPRLmMmXp1IYU0cILbSCcsqKn3dUjAgMBAAGjGDAW
MBQGA1UdEQQNMAuCCWxvY2FsaG9zdDANBgkqhkiG9w0BAQsFAAOCQAQEAiFqVRHMk
dRwTz07ilxBfkMhqniEifGMVnVLCj/4uoF1ons6dgi7wjwiDNwrgUCXE3wB3SHSD
AmuUzfrq/g+WWOMkBFra+cb2JBuFsOE6CiF4giDC8JcDdqeFVaKGbYVwgNTntSh0
Q/cbHFXNrCf11dQLInTEmdLeuvWml517vDXhoUuw25xIcM69NBvIe5IiSnTxF21d
NEhaxyynb0nXhIUeL+kYrecoCjmlwzh0BWzWpU6tQdCBaPYv709YuQww1lU7rT43
LBCz3Aeb8WmMm6vwJSWovrD/nLeLicBsEmV0rNomxIkR1FnNFQaBjk0F4gcJON0g
3RfglPNnGs/nIQ==
-----END CERTIFICATE-----

```

---

Listing 60: Corresponding X.509 certificate.