TLS 1.3 for strongSwan:
A Client-Side Prototype

Technical Documentation
Study Project, Spring Term 2020
Departement of Computer Science
University of Applied Sciences Rapperswil (HSR)
www.hsr.ch

Authors:
Pascal Knecht (pascal.knecht@hsr.ch, @ryu_foo)
Mélina Sieber (meline.sieber@hsr.ch, @MelineSieber)

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

Date: 29th May 2020
Contents

1. Abstract 6

2. Introduction 7
   2.1. Overview 7
   2.2. The Transport Layer Security Protocol (TLS) 8
   2.3. strongSwan and TLS 12
   2.4. Project Scope 14

3. Implementation 19
   3.1. Architecture Overview of libtls 19
   3.2. Design Choices 23
   3.3. HMAC-based Key Derivation Function 25
   3.4. Range-Based Version Handling 31
   3.5. Handshake and State Machine: Notable Changes 32
   3.6. Sequence Numbers 35
   3.7. Tests 36

4. Results 38

A. List of Abbreviations 41

B. Bibliography 42

C. Compile and Execution Instructions 44
   C.1. Set up Test Infrastructure with OpenSSL 44
   C.2. Set up strongSwan 45

D. Source Code 47

E. Test Data 49
   E.1. Local Connection: TLS 1.3 Client to TLS 1.3 Server 49
   E.2. External Connection: TLS 1.3 Client to TLS 1.3 Server 52
   E.3. Local Connection: TLS 1.2 Client to TLS 1.3 Server 54
   E.4. Local Connection: TLS 1.2 Client to TLS 1.2 Server 58
   E.5. Local Connection: TLS 1.3 Client to TLS 1.2 Server 61
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The location of TLS 1.3 and strongSwan in the OSI model.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Full handshake in TLS 1.2 with server authentication.</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Full handshake in TLS 1.3 with server authentication.</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Package diagram: strongSwan components that rely on a TLS stack</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Client-side state machine for a minimal viable product</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Server-side implementation</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Client-side authentication with certificate</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Class diagram of libtls at the start of the project</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Sequence diagram of libtls when the project started</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Each HKDF-Extract signals a one-way state transition</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>The state machine for our HKDF implementation with the four phases</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>UML class diagram of our current HKDF implementation</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>Basic concept of authenticated encryption with associated data ( AEAD )</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>A full TLS handshake with server authentication</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>Transitions from unencrypted to encrypted traffic in TLS 1.2 and 1.3</td>
<td>32</td>
</tr>
<tr>
<td>3.9</td>
<td>Successful TLS negotiation test cases</td>
<td>36</td>
</tr>
<tr>
<td>3.10</td>
<td>Screenshot of a handshake between a TLS 1.3 server and a strongSwan client</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Affected files</td>
<td>38</td>
</tr>
<tr>
<td>E.1</td>
<td>Wireshark: Handshake between a strongSwan TLS 1.3 client and an OpenSSL server supporting TLS 1.3</td>
<td>49</td>
</tr>
<tr>
<td>E.2</td>
<td>Wireshark: Handshake between a TLS 1.3 server and an external Google server</td>
<td>53</td>
</tr>
<tr>
<td>E.3</td>
<td>Wireshark: Handshake between a strongSwan TLS 1.2 client and an OpenSSL server supporting TLS 1.3</td>
<td>55</td>
</tr>
<tr>
<td>E.4</td>
<td>Wireshark: Handshake between a strongSwan TLS 1.2 client and an OpenSSL server supporting only TLS 1.2 in Wireshark</td>
<td>58</td>
</tr>
<tr>
<td>E.5</td>
<td>Wireshark: Handshake between a strongSwan TLS 1.3 client and an OpenSSL server supporting only TLS 1.2</td>
<td>61</td>
</tr>
</tbody>
</table>
Semester Project 2020

TLS 1.3 for strongSwan

Students: Pascal Knecht, Méline Sieber
Advisor: Prof. Dr. Andreas Steffen
Issue Date: Monday, February 17th 2020
Submission Date: Friday, May 29th 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.

The task for the semester project defined by this document is to extend the strongSwan libtls library with a minimal viable TLS 1.3 implementation.
Objectives

Mandatory:
- Getting acquainted with the existing TLS source code written in the C programming language and using some strongSwan-specific object-oriented macros as implemented by the strongSwan libtls library [1].
- Understanding the TLS 1.3 standard as defined by RFC 8446 [2] and identifying the basic differences and extensions relative to TLS 1.2.
- Implementation of a minimal viable TLS 1.3 client without client-side authentication.
- Negotiation of fallback to legacy TLS 1.x versions.
- Testing of minimal viable TLS 1.3 client against TLS 1.3 OpenSSL server.
- Interoperability of minimal viable TLS 1.3 client with TLS 1.3-enabled web sites (Google, Facebook, etc.).

Optional:
- Implementation of a minimal viable TLS 1.3 server
- Implementation of X.509 certificate-based TLS 1.3 client authentication

Links
[1] strongSwan libtls library, github source code repository
    https://github.com/strongswan/strongswan/tree/master/src/libtls

Rapperswil, February 17th 2020

Prof. Dr. Andreas Steffen
1. Abstract

**Introduction:** The Transport Layer Security protocol (TLS) 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 unmodified. The most widely used TLS version today is still version 1.2 from 2008 (RFC 5246), even though 1.3 has been released in 2018 (RFC 8446). The strongSwan project, which is maintained by the University of Applied Sciences Rapperswil (HSR), is an open-source IPsec implementation written in C. strongSwan features its own TLS stack in the library libtls. It enables client-authentication via various EAP authentication methods (TLS, TTLS, PEAP), which is used to establish an IKEv2 connection. However, libtls supports only TLS up to version 1.2.

**Objective:** This project aims to implement the client-side of TLS 1.3 in libtls and to update the TLS stack. The concrete goal is to successfully perform a minimal handshake and then exchange application data between a strongSwan client and a server running TLS version 1.3. As part of this minimal handshake, 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. This has considerably changed in the new version, which also implies that the handshake flow and state machine must be adapted too. Our scope includes three optional features: The server-side in a TLS handshake, client-side authentication and remaining non-mandatory extensions.

**Result:** The updated client implementation in libtls can successfully perform a minimal handshake with a server running TLS 1.3. It has been successfully and extensively tested with external TLS 1.3 servers such as those from Google or Facebook, but also with a local OpenSSL server. During implementation the new cryptographic mechanisms proved to be more difficult than originally anticipated. Especially the HKDF (HMAC-based Extract-and-Expand Key Derivation Function, RFC 5869) was an unexpected major challenge, especially since it is only marginally described in the TLS specification. Due to these difficulties, only the minimal handshake was implemented, the client-side authentication and the server-side was omitted.

For future work, the HKDF implementation needs attention: It is functional, but the code needs to be refactored. The features defined as optional, i.e. the server-side and client-side authentication, were not implemented due to time constraints. Still, they are relevant to the strongSwan project and will hopefully be completed as a Bachelor thesis.
2. Introduction

2.1. Overview

Rome wasn't built in a day, and the Transport Layer Security (TLS) in version 1.3 neither. TLS is the standard that secures connections between a server and a client and is just above layer four of the OSI model, the transport layer (see figure 2.1). The most popular TLS version 1.2 was defined in 2008 (RFC 5246), but only found broad adoption around 2014. In that same year, work on TLS 1.3 started. After several revisions, it was written down in RFC 8446 (2018), with significant departures from its predecessor. It took thus roughly ten years to introduce a new version of the TLS protocol. Rome took longer to build, but to be fair, TLS is no city. The strongSwan project spans a similar time frame and secures communications on networks as well. Its IPSec implementation authenticates and encrypts network traffic on layer three, the network layer. Originally, strongSwan was a 2005 fork of the discontinued FreeS/WAN project by John Gilmore. The project features its own TLS stack, yet only up to TLS version 1.2. The present work aims to implement RFC 8446 in strongSwan on the client-side and thus provides an update to the latest TLS version 1.3.

In the remaining sections of this chapter we shortly introduce the main features of TLS 1.3 and its major differences to its predecessor. We also describe the strongSwan project in greater detail and why there is a need for a TLS stack in an IPsec solution. Lastly, to comply with the time frame given, we must narrow down our goals to a manageable scope.

After the introduction, we turn to our implementation of RFC 8446. The strongSwan code is well-documented and follows its own coding style. However, architecture information is lacking, especially in the form of diagrams and illustrations. This encouraged us to document the existing architecture of the TLS stack code. We then justify our design decision that had a greater impact on the overall code. The next sections describe how new cryptographic processes are now part of the TLS stack in strongSwan and other notable changes to the existing code.

Lastly, we present our final product in the remaining chapter and how our design decisions and the overall construction process have influenced the outcome.

---

1 If not indicated otherwise, all illustrations are our own.

Sch17
Ste05
2.2. The 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.

Communication between client and server is achieved by exchanging messages of the Record Protocol, which in itself encapsulates the Handshake protocols, the Alert protocol, the ChangeCipherSpec protocol and the Application Data protocol. In order to establish a secure connection and transmit encrypted data, the protocol passes through three stages. In a first stage, a handshake establishes how client and server want to communicate with each other. This is the task of the Handshake protocol of TLS, which “authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material.” During the second stage, the actual data is sent, for example the content of a website. This, on the other hand, is the task of the Application Data protocol, which uses the parameters established during the handshake to protect the traffic. This third stage is marked by the termination of the connection by either one of the peers.

The project presented here focuses on the handshake process. Figure 2.2 shows a handshake in TLS version 1.2:

![Handshake in TLS 1.2 with server authentication](image)

Figure 2.2.: Full handshake in TLS 1.2 with server authentication.

In short, 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.

---

*Cf. Aum18 p. 236] and Fal12 p. 876].
Fal12 p. 877.
Res18 p. 6.
Res18 p. 6.
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 information in a `ClientKeyExchange` message. Note that the messages exchanged up to this point are still sent in plaintext.

If the handshake was successful so far, enough cryptographic material is available to encrypt the ensuing traffic. The client signals this state with a message from the `ChangeCipherSpec` protocol. A `Finished` message concludes the handshake. After the server has responded with the same messages, encrypted application data can flow between the two endpoints, sent using the `Record` protocol.

TLS 1.2 was an important step towards securing communication over the Internet, yet has its flaws and disadvantages. 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. The current version of TLS 1.3 presents a major overhaul of the 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 connection between client and server, has been greatly simplified in TLS 1.3. This becomes most apparent when we compare the state machines of the two TLS versions: The official specification does not list any state machine at all, with the various possible state machines leading to some messy graphics and faulty implementations. On the other hand, TLS 1.3 devotes a whole appendix chapter that lists the various possible state machines.

In comparison to figure 2.2, server and client exchange fewer messages:

![Handshake Diagram](image)

Figure 2.3.: Full handshake in TLS 1.3 with server authentication.

---

8 We presume a basic knowledge of TLS here and note only the most important aspects, and major differences and improvements in TLS 1.3 that are also relevant to our work. More detailed changes that became apparent during implementation are described in the following chapter. A good introduction to TLS 1.2 can be found in [Ris17].


10 [Beu17].

11 [Res18] p. 120.
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 eliminated. It has only been preserved to enable TLS 1.3 connections over middleboxes such as firewalls and intrusion detection systems and is not processed further. In addition to this speed-up, TLS 1.3 even allows for a zero-round-trip (0-RTT) if the client has previously connected to the server.

The other important improvement in figure 2.3: Data is encrypted much earlier in the whole client-server message exchange. The encryption happens right after the ServerHello, since the key material is earlier ready 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 the figure above 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 re-organized. This change is most visible in the list of cipher suites of which a client can offer a selection to the server. The page-long cipher suite combinations have been reduced to just five recommended cipher suites. Cipher suites in TLS 1.2 are of the following format, with an example just below:

\[
\text{TLS} \cdot [\text{key exchange}] \cdot [\text{authentication}] \cdot \text{with} \cdot [\text{cipher}] \cdot [\text{mac or prf}]
\]

\[
\text{TLS} \cdot \text{ECDHE} \cdot \text{RSA} \cdot \text{WITH} \cdot \text{AES} \cdot 128 \cdot \text{GCM} \cdot \text{SHA256}
\]

This is thus a cipher suite that uses ECDHE for the key exchange, RSA for authentication, 128bit-AES in GCM mode as bulk cipher to decrypt the application data, and SHA-256 as hash function. In TLS 1.3, there remain only five cipher suites using Authenticated Encryption with Associated Data (AEAD):

\[
\text{TLS} \cdot [\text{AEAD}] \cdot [\text{hash}]
\]

\[
\text{TLS} \cdot \text{AES} \cdot 128 \cdot \text{GCM} \cdot \text{SHA256}
\]

This is the 128bit AES-GCM authenticated encryption algorithm together with a SHA256-based pseudo-random function (PRF). The key exchange and signature algorithm field have been moved to their own respective extension. In addition, the recommended ciphers for key exchange all support forward secrecy. If an attacker gains the private key of a server or client, she is not able to decrypt past 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 in 2010. We discuss the consequences of the new AEAD cipher suites and the new HDKF in the chapter 3.3.

---

12 Res18, pp. 77, 140.
13 The IANA lists over more than 300 cipher suites: https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml, visited on 2020-04-23.
14 Res18, p. 133.
15 For a detailed description of AEAD see Aum18, p. 148.
16 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). Both concepts are explained in the implementation chapter.
17 Kra10.
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. Yet smaller providers have been rather slow in adopting the new TLS version as the authors note. We hope to speed up the process and at least lay the foundation for the strongSwan project for TLS 1.3 support.
2.3. strongSwan and TLS

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. 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. Since strongSwan version 4.5.0, charon supports EAP-TLS to mutually authenticate client and server with certificates.

The Extensible Authentication Protocol (EAP) 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. A Public Key Infrastructure (PKI) is required and must be able to provide a client certificate.

- **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.

- **Protected EAP (PEAP)** as developed by Cisco Systems, Microsoft and RSA Security 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.

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). These use cases are the main reason why a TLS stack is present in strongSwan.

The TLS stack has been explicitly written from scratch for strongSwan as the library libtls. TLS version 1.0, 1.1 and 1.2 and most of the associated cipher suites and cryptographic primitives are currently supported.

Figure 2.4 shows which components rely on this TLS stack:

---

1. Str18.
2. Str17.
3. Sim08.
4. Str15.
5. Wik.
7. Sta17, p. 165. Note that EAP-TTLS and PEAP do not need client certificates for authentication.
8. Sta17, p. 165.
11. San15.
Figure 2.4.: 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 libtls.

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.
2.4. Project Scope

TLS 1.3 is defined in an extensive RFC standard\textsuperscript{30} to achieve a minimal viable product (MVP), we mostly follow the compliance requirements as detailed in chapter 9 of the RFC.

2.4.1. Minimal Viable Product

The goal of this work is to implement the client-side components of a minimal TLS 1.3 handshake to a TLS 1.3-compliant server such as OpenSSL 1.1.1c.

This includes the mandatory new cryptographic computations, state machine, messages and extensions. The code to be written resides mainly in the strongSwan TLS library libtls. The code is written for strongSwan version 5.8.4.

Messages

Our minimal implementation uses the following messages:

- ClientHello
- HelloRetryRequest
- ServerHello
- EncryptedExtensions
- Certificate
- CertificateVerify
- Finished

Extensions

The RFC standard defines these mandatory extensions which must be implemented for a valid TLS 1.3 client\textsuperscript{31}

- server\_name(0): Defined in RFC 6066 and already implemented in strongSwan
- supported\_versions(43): New in TLS 1.3, chapter 4.2.1
- cookie(44): New in TLS 1.3, chapter 4.2.2
- signature\_algorithms(13) and signature\_algorithms\_cert(50): New in TLS 1.3, chapter 4.2.3
- supported\_groups(10), chapter 4.2.7
- key\_share(51): New in TLS 1.3, chapter 4.2.8

\textsuperscript{30} Res18
\textsuperscript{31} Res18 p. 103.
State Machine

The state machine below shows which part we implement to achieve a minimal viable product, drawn in green colour.

Figure 2.5.: Client-side state machine for a minimal viable product (green).

Legend:
green = minimal viable product
Can send app data

Res18, 120ff, The state machines shown are slightly simplified to improve readability.
2.4.2. Optional Features

There are three optional features that can be added to the MVP, listed in order of priority: server-side implementation, client-side authentication, and the remaining non-mandatory TLS 1.3 extensions.

Option 1: Server-Side Implementation

The existing TLS library in strongSwan also provides the server role of TLS 1.2 and plays a vital part in how strongSwan is used. Figure 2.6 shows the server-side state machine to our client-side MVP in orange colour.

![Server-side implementation (orange colour).](image-url)
Option 2: Client-Side Authentication with Certificate

EAP entails the concept of mutual authentication, where EAP-TLS provides the client-side authentication via a X.509 certificate that is used in a TLS connection. Figures 2.7 and 2.6 show the path through the client and server state machines in blue colour.

Figure 2.7.: Client-side authentication with certificate (blue color).
Option 3: Further TLS 1.3 Extensions

The following extensions are new in TLS 1.3, but not mandatory to be compliant.

- early_data(42)
- certificateAuthorities(47)
- oid_filters(48)
- post_handshake_auth(49)

2.4.3. Out of Scope

In order to keep the scope small, we defined these points to be out of scope:

- Middleboxes (for example reverse and forward proxies, firewalls, load balancers etc.)
- Pre-Shared Key (PSK) that allows a zero-roundtrip (0-RTT)
3. Implementation

In the following sections we describe how the minimal viable product was incorporated in the existing TLS stack of the strongSwan project. We start with a short description of the existing code base in libtls and explain our main design choice. The remaining sections show our major additions to and adaptations of the code base: The new HMAC-based key derivation function (HKDF) and the Authenticated Encryption with Associated Data (AEAD), as well as the new handshake and state machine. The chapter closes with a section on sequence numbers.

3.1. Architecture Overview of libtls

Our implementation extends the libtls-library of the strongSwan project. The library encompasses roughly 11'200 lines of code, spread over 17 .c- and 19 h-files. In order to make informed decisions about the implementation and to seamlessly integrate the new TLS stack, it was necessary to map and visualise the existing code.

3.1.1. Class Diagram

The existing code follows a special C-syntax that allows a pseudo-object-oriented coding style. The object-oriented code is achieved in the following way: Every .h- and corresponding .c-file is a class, e.g. the Crypto class corresponds to the files tls_crypto.c and tls_crypto.h. Header-files without a corresponding .c-file are treated as interfaces, e.g. the interface Handshake (tls_handshake.h) is implemented by the classes Peer and Server. Every class follows the same structure, using the Crypto-class as an example:

Private data of a class is indicated by the prefix private as in
struct private_tls_crypto_t. An object is instantiated using the create-function and a special macro called INIT, e.g.
tls_crypto_t *tls_crypto_create(tls_t *tls, tls_cache_t *cache). Since it is created on the heap, destroy() takes care of the memory management when the object is no longer needed.

The public methods of the Crypto class are of type struct tls_crypto_t and implemented using a macro called METHOD. It has the following schema

METHOD([class name], [method name], [return value], [private this],
       [list of arguments]...)

1 Measured using the Statistic plugin for IntelliJ CLion.
2 The project’s own coding guidelines have more detailed descriptions about the chosen style. The explanations here only describe what is necessary to understand the diagrams [Hut05, p. 56].
3 For the sake of simplicity we chose to only list the relevant parts of those methods in the diagram, i.e. name of the method, arguments and return value.
Diagram 3.1 shows how libtls was organised at the start of the project. Relationships between classes are indicated by an arrow. If an arrow points from class X to class Y, it means that class X has a reference of type Y in its private part. Thus the line tls_aead_t *aead_in in struct private_tls_crypto_t, which is the AEAD transformation for inbound traffic, uses something from the AEAD class, i.e. from tls_aead.c.

3.1.2. Sequence Diagram

The UML sequence diagram on page 22 illustrates the call order and object lifetime in a simple client-side TLS handshake situation.

The first call made by tls_test represents the libtls executable, which calls the function tls_socket_create of class Socket. Note this is a simplified diagram that emphasises the call order regarding the libtls library, removing the return arrows usually found in sequence diagrams. The vertical white beam represents the object life-time, which ends when the object’s destroy function is called, as explained in the previous section.
Figure 3.1: Class diagram of libtls at the start of the project.
Figure 3.2.: Sequence diagram of libtls when the project started.
3.2. Design Choices

Before starting out with our implementation, we faced two possible paths:

1. Write the new TLS stack from scratch.
2. Extend the current TLS stack and distinguish the version where needed.

As described in the introduction, the handshake in TLS 1.3 has been greatly simplified. TLS 1.3 does away with a lot of old baggage from previous versions, so a fresh implementation in new, separates files makes a lot of sense. In addition, the existing code is already highly complex, and an integration into the existing stack means adding even more complexity. On the other hand, this path implies that the team needs to thoroughly understand the current implementation also in the context of the whole strongSwan project, even though both team members have not worked with the code base before.

The second path reduces the need for the team to fully understand all the inner workings of the libtls library and how it connects to the rest of the code. It does not change the existing architecture and tries to apply changes and checks where needed. However, this alternative introduces many new version distinctions in the shape of if-else branches. Also, more complexity is the foe of good security, and a change in one place might introduce vulnerabilities in another or just break the code.

After careful deliberation, we chose the second path, in view of the new-to-us code and the limited time frame given. We therefore tried to work with the existing code where possible. This also has the advantage that it respects existing structures. New functions or files were only introduced where the new RFC deviated form the previous versions. In the code, this is mostly visible as this check:

```c
if (this->tls->get_version_max(this->tls) < TLS_1_3)
{
    /* existing code, TLS 1.2 */
}
else
{
    /* code for TLS 1.3 */
}
```

The maximum version is the highest version supported by the client and the version on which client and server have agreed.

Our initial decision led to many subtle and more obvious code changes scattered throughout the code base, mostly in tls.peer.c and tls.crypto.c. The most apparent change is visible in the list of cipher suites, which used to fill pages in TLS 1.2 and has been boiled down to just five in TLS 1.3. The list with the old and new cipher suites is located in the file tls_crypto.c as static suite_algs_t suite_algs[]. Initially, we added the five new cipher suites to this list, but they were all filtered out by the private method filter_unsupported_suites. In TLS 1.3, the cipher suite structure stays the same, but the key exchange and authentication fields have been turned into extensions. If we set them to default values only, they were filtered out. This is why we had to distinguish the new and old TLS versions in the cipher suite list and enhanced the existing suite_algs_t struct with a TLS version field.

*See section 2.2*
By using this new field we were able to apply only the filters relevant to the desired TLS version:

```c
typedef struct {
    tls_cipher_suite_t suite;
    key_type_t key;
    diffie_hellman_group_t dh;
    hash_algorithm_t hash;
    pseudo_random_function_t prf;
    integrity_algorithm_t mac;
    encryption_algorithm_t encr;
    size_t encr_size;
    tls_version_t tls_version; /* new field */
} suite_algs_t;
```

Further consequences of our design choices became apparent when we had to intertwine the new TLS 1.3 state machine with the existing one and establish a connection to send application data.\[^5\]

[^5]: See section 3.5
3.3. 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. TLS 1.3 on the other hand uses an HMAC-based Key Derivation Function (HKDF) to generate traffic keys. strongSwan does not bring a ready-to-use HKDF implementation the way TLS 1.3 requires. We had to add this functionality based on provided HKDF boilerplate code described in appendix D.0.2. This section describes the theoretical details of the HKDF in TLS 1.3 and our concrete implementation.

3.3.1. Theory

An HKDF has two fundamental functions: HKDF-Extract and HKDF-Expand. However, TLS 1.3 introduces two new additional functions, HKDF-Expand-Label and Derive-Secret. This results in these four functions:

- **HKDF-Extract** extracts a pseudo-random key (PRK) from the source key. 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 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-Extract-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: Relevant to our implementation are these labels:
    - "tls13 c hs traffic"

---

<table>
<thead>
<tr>
<th>Reference</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Res08</td>
<td>26.</td>
</tr>
<tr>
<td>Res18</td>
<td>95.</td>
</tr>
<tr>
<td>Kra10</td>
<td></td>
</tr>
</tbody>
</table>

1. The basic concepts of an HKDF are described in chapter 7.1, "Key Schedule", and chapter 7.3, "Traffic Key Calculation" [Res18 91ff].
2. Dan Boneh has an excellent video explaining HKDF in detail. See [Bon20].
4. For all eleven labels see the overview on page 93 [Res18].

25
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 are reduced to five, only two possible hashing algorithms currently remain: SHA-256 or SHA-384.

Figure 3.3 illustrates how the HKDF is used in TLS, and is directly drawn from the illustration in RFC 8446. 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.
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.

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

- Four PSK keys are derived from the Early Secret:
  - “tls13 ext binder”
  - “tls13 res binder”
  - “tls13 c e traffic”
  - “tls13 e exp master”
- Two handshake traffic secrets are derived from the Handshake Secret:
  - “tls13 c hs traffic”
  - “tls13 s hs traffic”
- The last four secrets are used for application traffic secrets and other use cases of TLS 1.3. They are derived from the Traffic Secret:
  - “tls13 c ap traffic”
  - “tls13 s ap traffic”
  - “tls13 exp master”
  - “tls13 res master”

There are a couple of labels that are used directly on HKDF-Expand-Label and can be called on all states of the HKDF state machine except phase 0:

- Provide actual keying material:
  - “tls13 key”
  - “tls13 iv”
- Generate the key to authenticate the handshake Finished messages:
  - “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.

---

15Res18, p. 93.
16Section 7.3 of Res18, p. 95.
17Section 4.4.4 of Res18, p. 71.
3.3.2. Implementation

The HKDF is used in different stages of the handshake process and functions like a state machine itself. We therefore called these internal states of the HKDF “phases”:

- Phase 0: the initial state
- Phase 1: generate pre-shared keys (not in our scope)
- Phase 2: generate handshake keys
- Phase 3: generate application data keys

Figure 3.4 illustrates the link between the HKDF in figure 3.3 and our phases.

Figure 3.4.: The state machine for our HKDF implementation with the four phases.

These two aspects – repeated use and internal states – encouraged us to encapsulate our whole implementation in an HKDF class. We designed the public interface of the HKDF class as visualised in figure 3.5.
The class is implemented in `tls_hkdf.h` and `tls_hkdf.c`. We verify its functionality with unit tests in `test_hkdf.c` based on static data and keying material. 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 ten labels defined in `enum tls_hkdf_labels_t`, a user sets the HKDF state machine into the appropriate state using the function `generate_secret`. Therefore, the label `derived` is not exposed to the caller and is solely used as internal label.

As one may notice from figure 3.4, 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 our implementation we considered this fact and encapsulated these steps into the function `generate_secret`. 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.

### 3.3.3. HKDF and AEAD

The idea behind Authenticated Encryption with Associated Data (AEAD) is to encrypt the payload of a message and additionally authenticate the plaintext packet headers used to route the packet. 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. However, TLS 1.3 uses AEAD somewhat differently, and one of the maintainers adapted the code to

---

18 We use the handshake bytes and keying material provided by *The New Illustrated TLS Connection* ([Dr]). The unit tests also give a good idea on how to the class is used in the code.

19 See also illustration 3.6. It is heavily inspired by Dan Bonehs video explaining authenticated encryption ([Bon15]).

20 This can also be recognised by the block cipher mode CCM and GCM respectively ChaCha20-Poly1305 for the stream cipher.

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:

- 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 `TLSCiphertext` structure and always sets the `TLSInnerPlaintext` structure content type to `Application Data`, the interface had to be modified. 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.

---

RES18, p. 81.
RES18, p. 83.
3.4. Range-Based Version Handling

Prior to TLS 1.3, each peer sent its supported TLS version in the handshake version field. This was a single value: A client offered its highest possible version as a two-byte value in its ClientHello message, for example 0x0303 for TLS 1.2. The server responded with its highest supported version that was equal or smaller to the client, also a two-byte value in the ServerHello message. If both peers supported this, the handshake continued with this agreed-on TLS version.

TLS 1.3 deprecates this mechanism in favour of a more extensible approach. The new protocol version negotiation is outsourced to the supported_versions TLS handshake extension. This allows to specify a range of supported versions instead of a single value. If a client supports TLS 1.0 up to TLS 1.3, it sends the four corresponding hex values with the most preferred listed first.

The original value field from earlier TLS versions has been preserved, however: A TLS-1.3-capable client sets the ClientHello.legacy_version field to 0x0303 (TLS 1.2) due to backward compatibility, but always sends the supported_versions extension along. This means also that the hex value of TLS 1.3 (0x0304) is never set in a ClientHello.legacy_version or ServerHello.legacy_version field.

In addition, the TLS Record header, which contains legacy_record_version, is set to the value of TLS 1.2 for all TLS 1.3 connections. The RFC makes one exception due to compatibility reasons and requires an implementation to set the record that contains a TLS 1.3 ClientHello message to the value of TLS 1.0.

To address this new negotiation mechanism, we replaced the current TLS version interface get_version in tls.h with the two methods get_version_min and get_version_max. The latter has a dual function: It announces the highest supported TLS version of a peer, but later in the handshake returns the TLS version on which the peers agreed. This new version range capability is then used to create the ClientHello message within tls_peer.c.

Due to the fact that we omitted the whole compatibility topic, our current implementation does not quite conform to the RFC. Our implementation sets the ClientHello.legacy_version correctly to 0x0303 but also uses this value for the legacy_record_version in the record value, where 0x0301 would be the correct value.

---

24 Res18, p. 139.
25 Res18, p. 79.
3.5. Handshake and State Machine: Notable Changes

Until client and server can exchange encrypted application data, they go through a handshake to agree on the specifics of the connection. As mentioned in the introduction, the RFC for the new TLS version lists state machines for server and client, information that was missing in the previous RFC for TLS 1.2. This section details how we incorporated these changes in handshake and state machine into the existing code.

In comparison to its earlier version, TLS 1.3 simplifies the handshake, removes a round-trip and some old messages:

- **Obsolete:** ServerHelloDone, ServerKeyExchange, ClientKeyExchange
- **New:** HelloRetryRequest, EncryptedExtensions

The other, less notable change is the transition to encrypted data. In TLS 1.2, the transition from plaintext traffic to encrypted application data is announced by the ChangeCipherSpec protocol message. It signals the peers that they are now both ready to transition to encrypted traffic. In the end, there are only two states: unencrypted and encrypted traffic.

On the other hand, TLS 1.3 has three states and no dedicated messages that signal a transition (see figure 3.8). The transition from state 0 to state 1 happens after the client receives the ServerHello message: At that point, the peers have agreed on the TLS version and the cipher suite for all the remaining handshake messages. All further messages are now encrypted with the handshake traffic secret – a notable change to TLS 1.2, which sent nearly all handshake messages in plaintext.

<table>
<thead>
<tr>
<th>State 0</th>
<th>State 1</th>
<th>State 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TLS 1.2</strong></td>
<td>plaintext handshake</td>
<td>encrypted application data</td>
</tr>
<tr>
<td><strong>TLS 1.3</strong></td>
<td>plaintext handshake</td>
<td>encrypted handshake</td>
</tr>
</tbody>
</table>

Figure 3.8.: Transitions from unencrypted to encrypted traffic in TLS 1.2 and 1.3.

---

26 See also figure 3.7.
27 The ServerFinished message is encrypted.
In our implementation, the handshake proceeds as follows:

**Client sends ClientHello:** The client generates the necessary keys specific to TLS 1.3 via the command `lib->crypto->create_db(lib->crypto, CURVE_25519)`\(^{28}\) It also sends a range of extensions including the supported TLS versions\(^{29}\).

**Client receives ServerHello:** After the Client has processed the message, it initialises the agreed-on cipher suite and generates the key material needed for the rest of the handshake. This happens by way of the new HKDF which is described in the previous section.

**Client receives ChangeCipherSpec:** This message still exists in TLS 1.3, but it is only there for middlebox compatibility, devoid of the original purpose. It can be sent or received nevertheless at any point after the ClientHello messages\(^{30}\) If the strongSwan TLS client receives a ChangeCipherSpec message, the state `STATE_CIPHERSPEC_RECEIVED` is set but nothing happens whatsoever.

**Client receives EncryptedExtension:** This is the first message encrypted with the handshake traffic secret generated by the HKDF. None of the extensions pertaining to this message are within our scope, so a skeleton was just implemented in a new method called `process_encrypted_extensions` in `tls_peer.c`.

**Client receives Certificate:** The message\(^{31}\) has two additions in comparison to TLS 1.2: A certificate request context is prepended to the list of certificates, and each certificate entry ends with extensions. Since we did not include the latter in our minimal viable product, the data is just read, but not further processed.

**Client receives CertificateVerify:** In TLS 1.2, it is used for client-side authentication only\(^{32}\) and the TLS 1.2 client in strongSwan is only able to send such a message. In TLS 1.3, this message is now mandatory for server-side authentication via certificate: “To prove that the server owns the server certificate (giving the certificate validity in this TLS session), it signs a hash of the handshake messages using the certificate’s private key. The signature can be proven valid by the client by using the certificate’s public key.”\(^{33}\) The new method `process_cert_verify` processes now such a message\(^{34}\).

**Client receives Finished:** The new Finished differs greatly from TLS 1.2. It uses the new HKDF and its output is of variable length, whereas in TLS 1.2 it had a fixed length of 12 bytes. A new public method `calculate_finished_tls13` that calculates the data to be verified by the client or sent to the server for verification, both by way of the HKDF.

**Client sends Finished:** This concludes the handshake\(^{35}\) and marks the transition to application data. The HKDF is now used to generate the application traffic keys to de- and encrypt

---

\(^{28}\) For an explanation of the chosen curve see the results chapter in 4.

\(^{29}\) Our implementation sends `server_name`, `supported_versions`, `signature_algorithms`, `supported_groups` (elliptic_curve extension in TLS 1.2), and `key_share`.

\(^{30}\) Res18, p. 77.

\(^{31}\) Res18, p. 64.

\(^{32}\) Res08, p. 62.

\(^{33}\) Annotation to “Server Certificate Verify” in [On].

\(^{34}\) On a side note, the RFC for TLS 1.3 does not comply to its own description of the transcript hash as defined earlier in section 4.4.1 of the RFC. For CertificateVerify, the transcript hash is `Transcript-Hash(Handshake Context, Certificate)`\(^{35}\) p. 69]. Certificate is ambiguous, since it is listed separately from Handshake Context: Is it material from the server certificate (its public key) or is it part of the transcript hash, i.e. the transcript hash including the Certificate message? It turns out that it is the latter.

\(^{35}\) Res18, p. 71.
application data. Note that in TLS 1.3, the Finished messages are exchanged in the exact opposite order as in TLS 1.2 (see also figure 3.7): The server sends the Finished message first, then the client sends its own. Again the HKDF is used with the “finished” label to compute the message.

**Client sends and receives Post-Handshake Messages:** After the actual handshake, client and server can still exchange Handshake messages. They are also encrypted with the application traffic key. Since our OpenSSL test server sent a NewSessionTicket message, we simply parsed it but did not further process its contents.

The messages above transition through a state machine that is implemented in the public methods build (outgoing traffic) and process (incoming traffic) in tls_peer.c As detailed in section 3.2, we made the conscious decision to distinguish between the TLS version within the code and not re-implement everything from scratch.

Initially, we did not distinguish the state machines, since so many states are similar to TLS 1.2. The more we processed in our implementation, the more we faced problems due to the rising complexity. Only when we clearly split the state machines by version did we achieve a bug-free implementation.

---

36 Res18, p. 73.
3.6. Sequence Numbers

Client and server separately maintain a sequence number of reading and writing records. If the sequence numbers are not initialised and incremented properly, the messages sent back and forth cannot be decrypted. In contrast to TLS 1.2, the nonces are initialised twice in TLS 1.3: Once when the unencrypted traffic switches to encrypted handshake traffic, and once when the encrypted handshake traffic switches to encrypted application data traffic.

In our implementation, the sequence number is first set to zero when the aead object is created after the ServerHello. Also note tls_aead_create_seq, which actually creates the newly implemented aead object:

```c
/* File: tls_crypto.c*/
static bool create_aead(private_tls_crypto_t *this, suite_algs_t *algs)
{
    if (this->tls->get_version_max(this->tls) < TLS_1_3)
    {
        this->aead_in = tls_aead_create_aead(algs->encr, algs->encr_size);
        this->aead_out = tls_aead_create_aead(algs->encr, algs->encr_size);
    }
    else
    {
        this->aead_in = tls_aead_create_seq(algs->encr, algs->encr_size);
        this->aead_out = tls_aead_create_seq(algs->encr, algs->encr_size);
    }

    /* call resets sequence numbers: */
    this->protection->set_cipher(this->protection, TRUE, this->aead_in);
    this->protection->set_cipher(this->protection, FALSE, this->aead_out);
}

/* File tls_protection.c */
METHOD(tls_protection_t, set_cipher, void, private_tls_protection_t *this, bool inbound, tls_aead_t *aead)
{
    if (inbound)
    {
        this->aead_in = aead;
        this->seq_in = 0; /* sequence number reset */
    }
    else
    {
        this->aead_out = aead;
        this->seq_out = 0; /* sequence number reset */
    }
}
```

The sequence number is reset to zero when the Finished message is sent by the client to conclude the handshake: As soon as the state STATE_FINISHED_SENT is set, a call to this->crypto->change_cipher resets the sequence numbers.

37 Res18 p. 82.
38 “Each sequence number is set to zero at the beginning of a connection and whenever the key is changed” [Res18 p. 82]. Since TLS is built on top of TCP, packet loss or disorder is not a problem.
39 See section 3.3.3.
3.7. Tests

3.7.1. Test Environment

The TLS library in strongSwan comes with a set of basic unit test suites. `test_socket.c` specifies six client-server tests, which test both peers for the TLS protocol version 1.0 to TLS 1.2. `testSuites.c` tests the cipher suite names. We left both test suites untouched as discussed with our advisors since `test_socket.c` is unable to complete because of our changes on the client-side TLS implementation. Further, we added `test_hkdf.c` to verify the functionality of our newly-implemented HKDF feature. 

Except for the HKDF implementation, we excluded memory leak detection due to time constraints. It has to be noted that we only implemented the client-side TLS 1.3 stack, so we were not able to write unit tests similar to `test_socket.c` to automatically verify the functionality of our implementation. We had to test everything semi-automated with self-written shell scripts. The basic test setup consists of an OpenSSL server as described in appendix C and the freshly compiled `tls_test` binary.

We also manually tested the functionality of our TLS 1.3 client with servers on the Internet, e.g., www.google.com on port 443 as described in appendix E. It has to be mentioned that this approach is very error-prone, because the server certificate validation is non-deterministic. `tls_test` requires the `--cert` parameter to point to the server’s public key to verify authenticity of the connection. Since Google runs many servers with different server certificates we were not able to establish a connection with every attempt.

In table 3.9 we provide an overview of our semi-automated test cases for successful TLS version negotiation of various combinations. Test data of all our test cases can be found in appendix E.

<table>
<thead>
<tr>
<th>Client max TLS version</th>
<th>Server max TLS version</th>
<th>Negotiated TLS version</th>
<th>Test Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS 1.3</td>
<td>TLS 1.3</td>
<td>TLS 1.3</td>
<td>✓</td>
</tr>
<tr>
<td>TLS 1.2</td>
<td>TLS 1.3</td>
<td>TLS 1.2</td>
<td>✓</td>
</tr>
<tr>
<td>TLS 1.2</td>
<td>TLS 1.2</td>
<td>TLS 1.2</td>
<td>✓</td>
</tr>
<tr>
<td>TLS 1.3</td>
<td>TLS 1.2</td>
<td>TLS 1.2</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 3.9.: Successful TLS negotiation test cases.

3.7.2. Successful Handshake

On the next page we see a successful TLS 1.3 handshake between an OpenSSL server (left) and a strongSwan client (right). On the client side, the list of offered ciphers suites is visible at the top, albeit cut-off for brevity.

---


41 This happens because on a first connection, the server certificate from www.google.com is downloaded via a browser, and in a second step our client uses the certificate to connect to www.google.com. It is not guaranteed that the client then connects to the same server with the same certificate as in the first step.
Figure 3.10.: Screenshot of a handshake between a TLS 1.3 server (left) and a strongSwan client (right, 2020-05-21).
4. Results

The project entailed complex code, the terseness of an RFC and cryptographic operations which had to be understood and implemented. It was a wise decision to define modest goals and everything else as optional. In the end, we were able to implement the minimal viable product: We are able to successfully establish a TLS 1.3 connection between a client and a server and thus wrote the client-side of a TLS 1.3 connection. Figure 4.1 shows all files that were affected by our project, where green color indicates new files. Even if tls_aead_seq.c was added, its code affected all other aead files.

TLS 1.3 uses the new HKDF scheme to derive cryptographic keys. We had to implement this functionality while drawing on existing cryptographic primitives in strongSwan. The newly created implementation works fine from a functional perspective, unit tests with known traffic data and keying material are successful. Since RFC 8446 scatters explanations of the HKDF’s application throughout the text, our implementation of the HKDF grew organically along as we worked our way through the handshake. It must be left to future work to refactor the existing code and introduce a more simple and complete way to generate key material via the HKDF.

During the integration of our HKDF into the client-side handshake, it became clear that our initial concept of an HKDF implementation was not ideal and has some flaws. We therefore suggest the following implementation that is more robust and easier-to-use.

The public method generate_secret controls the flow of the internal state machine by the label it is called with. The method returns the secrets Early Secret, Handshake Secret and Master Secret. However, they are never used directly to derive concrete secrets. The code can therefore be improved as follows:

- Remove return value parameter in generate_secret, since the secrets are never used. This would simplify the API to callers.
- Remove the whole function and control the internal state machine flow directly with the other methods. This would reduce the amount of

---

1See also section 3.3.3.
public methods, but would lead to additional parameters for the functions `derive_key`, `derive_iv` and `derive_finished`.

The public method `derive_key` and `derive_iv` may be refactored to one method to simplify the interface even more. The public method `derive_finished` uses the same private function to compute its secret as `derive_key` and `derive_iv` and has no validity check if it is in phase 3. This could be improved by guaranteeing that the function only provides a secret when it is called in the correct state.

The above list is a set of (incomplete) suggestions which may not work well with each other and must be reviewed with the maintainers in more detail before they are implemented.

We did not implement the server side (option 1) and neither the client-side authentication with a certificate (option 2). What prolonged our implementation process were the HKDF itself, the complexities of extending the existing state machine and the final transition to an established connection. Future work must implement the two missing options that are relevant to strongSwan – since the server-side is still missing, our implementation only works with an (external) TLS 1.3-server.

In section 2.4.1 we defined all mandatory extensions that our client should support. In the end, we implemented all except one, the `cookie` extension, after the maintainers agreed that it was unneeded in the strongSwan project. The `HelloRetryRequest` message, which is also listed in our scope, was not implemented.

Regarding the cryptographic primitives, we added support for all new five cipher suites defined in RFC 8446. Our HKDF implementation supports both hashing algorithms. As key exchange method we currently only support Elliptic Curve Diffie-Hellman (ECDHE) with curve X25519. The reason for this is how key exchange mechanisms differ between TLS versions. TLS 1.2 and prior used information from the `ClientKeyExchange` message to calculate key material, TLS 1.3 calculates the client key material at the beginning and then sends it in the `ClientHello` message, thus in the first message of the client to the server. In addition, older TLS implementations only calculated one key pair using a negotiated algorithm. A TLS 1.3 client initially sends its key material generated with different algorithms and simply hopes that the server supports one of these. Therefore, other TLS 1.3 implementations usually calculate two key pairs. The existing code base currently does not offer a uniform way to calculate and store key exchange material from multiple algorithms. Because of this, we decided to support curve X25519 only.

As signature hash algorithms, we send these six in the `signature_algorithms` extension:

- `ecdsa_secp256r1_sha256` (0x0403)
- `ecdsa_secp384r1_sha384` (0x0503)

---

2 See also section 2.4.2
3 See appendix C on how to generate the key material and setup an OpenSSL server to see our code in action.
4 RFC 8446 specifies SHA-256 and SHA-384
6 Mozilla Firefox for example calculates and offers key exchange for secp256r1 and X25519, and the server picks its preference of the two.
7 According to the RFC, a TLS 1.3-compliant application must support secp256r1 and should support X25519. We chose the latter due to personal preferences although secp256r1 is also supported by strongSwan and should work as well, though we never actually tested this.
This is a subset of all the definitions on page 42 in RFC 8446. The implementation of the extension is dynamic, which means that the amount of supported signature hashes varies depending on how the compilation flags have been set.

In section 3.2 we decided to do an in-place TLS version distinction rather than write everything from scratch. As expected, it added complexity to the current code, and it is not easily to distinguish which code accounts for which TLS version. It may be another task to refactor the code to the point that files distinguish the TLS versions and not if-else branches.

Finally, this study project was a project like no other: Even though we faced a global pandemic thanks to COVID-19 with drastic changes to our daily lives, we are proud to have achieved our main goal nevertheless. Even though we did not build Rome in a day, we at least built a client-side TLS 1.3 stack.
A. List of Abbreviations

**AEAD** Authenticated Encryption with Associated Data  
**CA** Certificate Authority  
**CSR** Certificate Signing Request  
**EAP** Extensible Authentication Protocol  
**EAP-PEAP** EAP with Protected Extensible Authentication Protocol  
**EAP-TLS** EAP with Transport Layer Security  
**EAP-TTLS** EAP with Tunneled Transport Layer Security  
**GCM** Galois/Counter Mode  
**HKDF** HMAC-based Extract-and-Expand Key Derivation Function (RFC 5869)  
**ECDHE** Elliptic-Curve Diffie-Hellman Ephemeral  
**HMAC** Keyed-Hash Message Authentication Code  
**IKE** Internet Key Exchange  
**IKM** Input Key Material  
**IV** Initialisation Vector  
**MVP** Minimal Viable Product  
**OKM** Output Key Material  
**OSI** Open Systems Interconnection Model  
**PKI** 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  
**RTT** Round-Trip Time  
**SA** Security Association  
**TLS** Transport Layer Security
B. Bibliography


C. Compile and Execution Instructions

C.1. Set up Test Infrastructure with OpenSSL

We assume an OpenSSL version is installed, at least version 1.1.1 (September 2018).

(1) Generate a self-signed certificate authority (CA) with password 1337 (required). The certificate authority can be created with a certificate signing request (CSR, step 2.1) or without (step 2.2).

# 1. create key
$ openssl ecparam -genkey -name secp256r1 | openssl ec -out ca.key -aes128

# 2.1a create csr
$ openssl req -new -key ca.key -out ca.csr

# 2.1b sign ca certificate
$ openssl x509 -req -days 365 -in ca.csr -signkey ca.key -out ca.crt

# 2.2 self-signed ca without a csr
$ openssl req -new-x509 -days 365 -key ca.key -out ca.crt

(2) Server: generate a strong private key with password 7331:

$ openssl ecparam -genkey -name secp256r1 | openssl ec -out server.key -aes128

(3) Create a Certificate Signing Request (CSR) and send it to a CA, which is the one created in (1):

$ openssl req -new -sha256 -key server.key -out server.csr

(4) Sign the certificate:

$ openssl x509 -req -CAcreateserial -in server.csr -sha256 -CA ca.crt -CAkey ca.key -out server.crt

In the end, the following files exist in order of creation:

- Client side: ca.key, (ca.csr,) ca.crt
- Server side: server.key, server.csr, server.crt

Run the OpenSSL-server with password 7331:

$ openssl s_server -accept localhost:8443 -key server.key -cert server.crt -debug -pass pass:7331 -keylogfile keylogs
The flag `-keylogfile keylogs` writes all the keys to a file called `keylogs`. It allows Wireshark to decrypt sniffed traffic.

Run the strongSwan client:

```
$ ../strongswan-root/scripts/tls_test --connect localhost --port 8443
"--cert server.crt --debug 2
```

### C.2. Set up strongSwan

The most simple steps to set up strongSwan for our purpose is described in the following steps. Note that EAP-TLS and EAP-TTLS have to be explicitly enabled. Also the OpenSSL back-end to support most of the cryptographic primitives has to be configured explicitly.

```
$ git clone https://git.strongswan.org/strongswan.git
$ cd strongswan/
$ ./autogen.sh
$ make clean
$ ./configure --enable-eap-tls --enable-eap-ttls --enable-openssl
$ make
$ make install
```

Verify functionality by running all unit tests:

```
$ make check
```

The client can then be started by the following command:

```
$ ./scripts/tls_test
```

If only required plugins should be enabled, one can use the following configuration options:

```
$ ./configure --disable-aes --disable-des --disable-rc2 --disable-sha2
"--disable-sha1 --disable-md5 --disable-pgp --disable-dnskey
"--disable-sshkey --disable-gmp --disable-xcbc --disable-cmac
"--disable-hmac --disable-random --disable-pkcs1 --disable-pkcs7
"--disable-pkcs8 --disable-pkcs12 --disable-attr
"--disable-kernel-netlink --disable-stroke --disable-vici
"--disable-updown --disable-xauth-generic --disable-counters
"--disable-resolve --disable-socket-default --disable-nonce
"--disable-x509 --disable-revocation --disable-fips-prf --disable-drbg
"--disable-constraints --disable-pubkey --disable-pem
"--disable-curve25519 --enable-eap-tls --enable-eap-ttls
"--enable-openssl
```

Initially we tried to use valgrind to manually find memory leaks. Our advisors pointed out the configuration flag `--enable-leak-detective` which provides the same validation constraints also used by the automated unit tests. We strongly recommend to always set this option while working on the strongSwan project. Now, when running unit tests by `make check`, memory leaks are checked as well.

---

1 Via the menu points `Edit -> Preferences -> (Pre)-Master-Secret log filename`. 

45
Temporarily disabling memory leak detection can be achieved by setting the flag LEAK_DETECTIVE_DISABLE. The following command also sets the flag TESTS_VERBOSE to print more verbose output to stdout.

$ make check TESTS_VERBOSE=2 LEAK_DETECTIVE_DISABLE=0
D. Source Code

D.0.1. Code-Repository

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

https://github.com/bytinbit/strongswan

Interesting branches are:

- `sa-dev-libtls`: the final code
- `feature-mvp`: complete commit history until code freeze
- `feature-hkdf`: complete commit history of the HKDF implementation until code freeze

D.0.2. HKDF Boilerplate Code

The strongSwan maintainer Tobias Brunner\(^1\) provided the boilerplate code for an HKDF, implemented in strongSwan’s code base. The implementation conforms to RFC 5869\(^2\)

```c
#include <crypto/prf_plus.h>

prf_t *prf;
prf_plus_t *prf_plus;
chunk_t salt, IKM, PRK, info, OKM;
size_t L;

prf = lib->crypto->create_prf(lib->crypto, PRF_HMAC_SHA2_256);

/* HKDF-Extract(salt, IKM) -> PRK */
salt = chunk_from_chars(
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c,
);  
IKM = chunk_from_chars(
    0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b,
    0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b,
    0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b,
);  
prf->set_key(prf, salt);
prf->allocate_bytes(prf, IKM, &PRK);
```

\(^1\)Email from 26.3.2020
\(^2\)For the implementation in TLS 1.3, see section 3.3
DBG1(DBG_APP, "===%B", &PRK);

/* HKDF-Expand(PRK, info, L) -> OKM */
info = chunk_from_chars(
    0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
    0xf8, 0xf9,
);
L = 42;
prf->set_key(prf, PRK);
prf_plus = prf_plus_create(prf, TRUE, info);
prf_plus->allocate_bytes(prf_plus, L, &OKM);
DBG1(DBG_APP, "===%B", &OKM);

chunk_clear(&PRK);
chunk_clear(&OKM);
prf_plus->destroy(prf_plus);
prf->destroy(prf);
E. Test Data

E.1. Local Connection: TLS 1.3 Client to TLS 1.3 Server

A successful TLS 1.3 handshake, server side

openssl s_server -accept www.test.local:8443 -key tls_ecdsa/server.key -cert tls_ecdsa/server.crt -debug -pass pass:7331 -keylogfile output/keylogs

Using default temp DH parameters

```
read from 0x5562d4480b40 [0x5562d448b753] (5 bytes => 5 (0x5))
0000 - 16 03 03 01 01 0000 - 01 00 00 fd 03 03 5e ce-6b 14 47 c6 c0 4a 5c b1 .........
0010 - 07 ad bb 78 8c 7e ed 65-f0 ea 54 0d cd 85 f0 fa ...x..e..T.....
0020 - 3f ae 7d 56 02 91 00 00-5c 13 01 13 02 13 03 13 ...".....A.
0030 - ba 00 84 00 c0 08 c0-12 00 0a c0 06 c0 10 00 ............
0040 -
```

Figure E.1.: Screenshot of a handshake between a strongSwan TLS 1.3 client and an OpenSSL server supporting TLS 1.3 in Wireshark (2020-05-27).
Secure Renegotiation IS NOT supported

read from 0x5562d4480b40 [0x5562d448c753] (5 bytes => 5 (0x5))
0000 - 17 03 03 00 12 .

read from 0x5562d4480b40 [0x5562d448c758] (18 bytes => 18 (0x12))
0000 - 52 55 ec 7f bb 6a 9e 0b-ec 5b ee ca 27 e6 b8 3c RU...j...[...
0010 - ...

read from 0x5562d4480b40 [0x5562d448c753] (5 bytes => 5 (0x5))
0000 - 17 03 03 00 16 ...

read from 0x5562d4480b40 [0x5562d448c758] (22 bytes => 22 (0x16))
0000 - e8 e3 a3 0e 9f 81 af 95-cf b0 95 dd 1e ba 41 86 ............A.
0010 - 95 8f 96 bf e2 ......

ping pong

write to 0x5562d4480b40 [0x5562d44908a3] (27 bytes => 27 (0x1B))
0000 - 17 03 03 00 16 b9 fd 96-57 c6 54 f3 c4 e0 a2 90 ........W.T......
0010 - 11 da f3 33 f2 d0 32 7e-c4 a5 f2 ...3..2...

read from 0x5562d4480b40 [0x5562d448c753] (5 bytes => 0 (0x0))
ERROR
shutting down SSL
CONNECTION CLOSED

A successful TLS 1.3 handshake, client side

scripts/tls˙test --connect www.test.local --port 8443 --cert
↪ /home/pascal/Documents/Bildung/th/hsr/sem6/SA/sa-strongswan-doku/
↪ scripts/server-client/tls˙ecdsa/server.crt

negotiated TLS 1.3 using suite TLS˙AES˙128˙GCM˙SHA256
received TLS server certificate 'C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local'
no issuer certificate found for "C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local"
issure is "C=AU, ST=Some-State, O=Internet Widgits Pty Ltd"
using trusted certificate "C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local"

ping pong
~C

E.2. External Connection: TLS 1.3 Client to TLS 1.3 Server

A successful TLS 1.3 handshake with a Google server

$ ../../../strongswan˙code/scripts/tls˙test --connect www.google.com --port 443 --debug 2
↪ /home/pascal/Documents/Bildung/th/hsr/sem6/SA/sa-strongswan-doku/

46 supported TLS cipher suites:
TLS˙AES˙128˙GCM˙SHA256
TLS˙AES˙256˙GCM˙SHA384
TLS˙CHACHA20˙POLY1305˙SHA256
TLS˙AES˙128˙CHMAC˙SHA256
TLS˙ECDHE˙ECDSA˙WITH˙AES˙128˙CBC˙SHA
TLS˙ECDHE˙ECDSA˙WITH˙AES˙128˙CBC˙SHA256
TLS˙ECDHE˙ECDSA˙WITH˙AES˙256˙CBC˙SHA
TLS˙ECDHE˙ECDSA˙WITH˙AES˙256˙CBC˙SHA384
TLS˙ECDHE˙ECDSA˙WITH˙AES˙128˙GCM˙SHA256
TLS˙ECDHE˙RSA˙WITH˙AES˙128˙CBC˙SHA
TLS˙ECDHE˙RSA˙WITH˙AES˙128˙CBC˙SHA256
### Figure E.2.: Screenshot of a handshake between a TLS 1.3 server and an external Google server as shown in Wireshark (2020-05-23).

| SSL/TLS cipher suite | Time (ms) | Source (IP) | Destination (IP) | Protocol | Length (bytes) | Flags
|----------------------|-----------|-------------|------------------|----------|----------------|-------|
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA | 128.321.2.229 | 172.251.48.80 | 172.251.48.80 | TCP | 1552 | C
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA | 128.321.2.229 | 172.251.48.80 | 172.251.48.80 | TCP | 1552 | C
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA | 128.321.2.229 | 172.251.48.80 | 172.251.48.80 | TCP | 1552 | C
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA | 128.321.2.229 | 172.251.48.80 | 172.251.48.80 | TCP | 1552 | C
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA | 128.321.2.229 | 172.251.48.80 | 172.251.48.80 | TCP | 1552 | C
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA | 128.321.2.229 | 172.251.48.80 | 172.251.48.80 | TCP | 1552 | C
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA | 128.321.2.229 | 172.251.48.80 | 172.251.48.80 | TCP | 1552 | C
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA | 128.321.2.229 | 172.251.48.80 | 172.251.48.80 | TCP | 1552 | C
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA | 128.321.2.229 | 172.251.48.80 | 172.251.48.80 | TCP | 1552 | C
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA | 128.321.2.229 | 172.251.48.80 | 172.251.48.80 | TCP | 1552 | C

- **SSL/TLS cipher suite**: Various SSL/TLS cipher suites are exchanged between the two endpoints.
- **Time (ms)**: The time taken for each exchange is shown.
- **Source (IP)**: The IP address of the server initiating the handshake.
- **Destination (IP)**: The IP address of the server being contacted.
- **Protocol**: TCP is used for all connections.
- **Length (bytes)**: The length of each packet exchanged.
- **Flags**: Flags indicate the nature of the packet, such as the handshake type.

This screenshot illustrates a typical TLS handshake, where both servers exchange cipher suites and establish a secure connection.
E.3. Local Connection: TLS 1.2 Client to TLS 1.3 Server

A successful TLS 1.2 handshake, server side

openssl s_server -accept www.test.local:8443 -key tls_ecdsa/server.key -cert tls_ecdsa/server.crt -debug -pass pass:7331 -keylogfile output/keylogs

Using default temp DH parameters
Figure E.3.: Screenshot of a handshake between a strongSwan TLS 1.2 client and an OpenSSL server supporting TLS 1.3 in Wireshark (2020-05-27).

ACCEPT

read from 0x55f62dbb5b40 [0x55f62dbc1753] (5 bytes => 5 (0x5))
0000 - 16 03 03 00 fd
0000 - 01 00 00 f9 03 03 5e ce-67 13 46 16 ed 3a 5c 42 ......^g.F.:\B
0010 - 24 0b 81 db cd 61 50 c4-72 13 46 16 ed 3a 5c 42 ......^g.F.:\B
0020 - 30 04 13 03 c0 9e 00 9f 00 9e 00 9f 00 00-5c 13 01 13 02 03 04 05 ......\...\.
0030 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0040 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0050 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0060 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0070 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0080 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0090 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00a0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00b0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00c0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00d0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00e0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00f0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
write to 0x55f62dbb5b40 [0x55f62dbc1753] (715 bytes => 715 (0x2CB))
0000 - 16 03 03 00 fd
0000 - 01 00 00 f9 03 03 5e ce-67 13 46 16 ed 3a 5c 42 ......^g.F.:\B
0010 - 24 0b 81 db cd 61 50 c4-72 13 46 16 ed 3a 5c 42 ......^g.F.:\B
0020 - 30 04 13 03 c0 9e 00 9f 00 9e 00 9f 00 00-5c 13 01 13 02 03 04 05 ......\...\.
0030 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0040 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0050 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0060 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0070 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0080 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
0090 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00a0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00b0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00c0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00d0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00e0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...
00f0 - 39 00 6b 00 9e 00 9f 00 45 00 be 00 88 00 c4 00 9.k.....E...

A successful TLS 1.2 handshake, client side

scripts/tls_test --connect www.test.local --port 8443 --cert
↪ /home/pascal/Documents/Bildung/Fh/hsr/sem6/SA/ea-strongswan-doku/scripts/
↪ server-client/tls_ecdsa/server.crt

negotiated TLS 1.2 using suite TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
received TLS server certificate
'C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local'
no issuer certificate found for "C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local"
Figure E.4.: Screenshot of a handshake between a strongSwan TLS 1.2 client and an OpenSSL server supporting only TLS 1.2 in Wireshark (2020-05-27).

issuer is "C=AU, ST=Some-State, O=Internet Widgits Pty Ltd"
using trusted certificate "C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local"

ping pong

E.4. Local Connection: TLS 1.2 Client to TLS 1.2 Server

A successful TLS 1.2 handshake, server side

openssl sserver -accept www.test.local:8443 -key tls_ecdsa/server.key -cert tls_ecdsa/server.crt -debug -pass pass:7331 -keylogfile output/keylogs -tls1_2
Using default temp DH parameters
ACCEPT
read from 0x55a44e664b40 [0x55a44e670753] (5 bytes => 5 (0x5))
0000 -  16 03 03 00 0d 00 00 00 00 00 00 00 00 00 00 00
0010 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0090 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00a0 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00b0 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00c0 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00d0 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00e0 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00f0 -  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

58
0040 - 63 63 84 66 03 65
read from 0x55a44e664b40 [0x55a44e670753] (5 bytes => 5 (0x5))
0000 - 14 03 00 01
read from 0x55a44e664b40 [0x55a44e670758] (1 byte => 1 (0x1))
0000 - 01
read from 0x55a44e664b40 [0x55a44e670753] (5 bytes => 5 (0x5))
0000 - 16 03 00 40
write to 0x55a44e664b40 [0x55a44e679a20] (75 bytes => 75 (0x4B))
0000 - 14 03 00 01 01 16 03 03 00 40 ed b1 87 09 4a
0010 - 91 7e 6d 1f 73 3a 8f 04 fd 47 6e 88 3b f4 a1 75
0020 - 3f 86 2b 81 1a 7f 04 0d 8d 04 0d 7f 04 0d 7f
0030 - 04 0d 7f 04 0d 7f
write to 0x55a44e664b40 [0x55a44e679a20] (75 bytes => 75 (0x4B))
0000 - 17 03 03 00 30
read from 0x55a44e664b40 [0x55a44e670753] (5 bytes => 5 (0x5))
0000 - 17 03 03 00 16 03 03 00 40 ed b1 87 09 4a
0010 - 91 7e 6d 1f 73 3a 8f 04 fd 47 6e 88 3b f4 a1 75
0020 - 3f 86 2b 81 1a 7f 04 0d 8d 04 0d 7f 04 0d 7f
0030 - 04 0d 7f 04 0d 7f
read from 0x55a44e664b40 [0x55a44e670758] (48 bytes => 48 (0x30))
0000 - c6 91 87 3a 2a d0 d5 31 30 7e 00 71 07 ad 38 76 44 9e e8 e8 50 0f 04
0010 - .*..*O.K.g.
0010 - 60 68 29 18 21 4a aa b5 a0 e9 23 65 d2 37 4e df
0020 - 90 8d 8a 9e 28 33 23 f8 3d f6 01 8d 66 02 4a 86
ping pong
write to 0x55a44e664b40 [0x55a44e6748a3] (53 bytes => 53 (0x35))
0000 - 17 03 00 30 0f 6f 01 98 0c 0d 64 85 8d 03 69 d6
0010 - fc e6 cd 45 fd ba 89 74 c8 7f 81 29 66 38 cb 49
0020 - E...t...f8.I

-----BEGIN SSL SESSION PARAMETERS-----

MHUCAQECAgMDBALACQQgUaQ/w/mPwa14ydHtbIAxIo1oBxrqDrtzVtSFZ5XlP

Shared ciphers:TLS˙AES˙128˙GCM˙SHA256:TLS˙AES˙256˙GCM˙SHA384:
ECDHE-ECDSA-AES128-SHA: ECDHE-ECDSA-AES128-SHA256:
ECDHE-ECDSA-AES128-SHA384: ECDHE-ECDSA-AES256-SHA:
ECDHE-ECDSA-AES256-SHA256: ECDHE-ECDSA-AES256-SHA384:
ECDHE-RSA-AES128-SHA: ECDHE-RSA-AES128-SHA256:
ECDHE-RSA-AES128-SHA384: ECDHE-RSA-AES256-SHA:
ECDHE-RSA-AES256-SHA256: ECDHE-RSA-AES256-SHA384:
DHE-RSA-AES128-SHA: DHE-RSA-AES128-SHA256:
DHE-RSA-AES128-SHA384: DHE-RSA-AES256-SHA:
AES128-GCM-SHA256: AES128-GCM-SHA384:
Signature Algorithms: ECDSA+SHA256: ECDSA+SHA384:
Supported Elliptic Curve Point Formats: uncompressed
Supported Elliptic Groups: P-256: P-384: P-521:
CIPHER is ECDHE-ECDSA-AES128-SHA

Shared Signature Algorithms: ECDSA+SHA256: ECDSA+SHA384:
Supported Elliptic Groups: P-256: P-384: P-521:
Shared elliptic groups: P-256: P-384: P-521
CIPHER is ECDHE-ECDSA-AES128-SHA

Secure Renegotiation IS NOT supported
read from 0x55a44e664b40 [0x55a44e670753] (5 bytes => 5 (0x5))
0000 - 17 03 00 30
read from 0x55a44e664b40 [0x55a44e670758] (48 bytes => 48 (0x30))
0000 - c6 91 87 3a 2a d0 d5 31 30 7e 00 71 07 ad 38 76 44 9e e8 e8 50 0f 04
0010 - .*..*O.K.g.
0010 - 60 68 29 18 21 4a aa b5 a0 e9 23 65 d2 37 4e df
0020 - 90 8d 8a 9e 28 33 23 f8 3d f6 01 8d 66 02 4a 86

60
A successful TLS 1.2 handshake, client side

```
scripts/tls˙test --connect www.test.local --port 8443 --cert
 ↪ server-client/tls˙ecdsa/server.crt
```

negotiated TLS 1.2 using suite TLS˙ECDHE˙ECDSA˙WITH˙AES˙128˙CBC˙SHA

received TLS server certificate 'C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local'
no issuer certificate found for "C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local"
issuer is "C=AU, ST=Some-State, O=Internet Widgits Pty Ltd"

using trusted certificate "C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local"
ping pong ~

**E.5. Local Connection: TLS 1.3 Client to TLS 1.2 Server**

A successful TLS 1.2 handshake, server side
openssl s server -accept www.test.local:8443 -key tls_ecdsa/server.key -cert tls_ecdsa/server.crt -debug -pass pass:7331 -keylogfile output/keylogs -tls1_2

Using default temp DH parameters

ACCEPT

read from 0x564024a33b40 [0x564024a3f753] (5 bytes => 5 (0x5))
0000 - 16 03 03 01 01

read from 0x564024a33b40 [0x564024a3f758] (257 bytes => 257 (0x101))
0000 - 01 00 00 fd 03 03 5e ce-60 c3 9e 8f 48 3e 08 77 .

write to 0x564024a33b40 [0x564024a48a20] (715 bytes => 715 (0x2CB))
0000 - 16 03 03 00 54 02 00 00-50 03 03 e2 20 32 e3 89 ....T...P... 2..

write to 0x564024a33b40 [0x564024a48a20] (715 bytes => 715 (0x2CB))
0000 - 16 03 03 00 54 02 00 00-50 03 03 e2 20 32 e3 89 ....T...P... 2..

0010 - 1b .
A successful TLS 1.2 handshake, client side

negotiated TLS 1.2 using suite TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA

received TLS server certificate 'C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local'
no issuer certificate found for "C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local"
issuver is "C=AU, ST=Some-State, O=Internet Widgits Pty Ltd"

using trusted certificate "C=CN, ST=GD, L=SZ, O=Magrathea, Inc., CN=www.test.local"
ping
pong

^C