OPTLS Revisited

Wouter Kuhnen
Advisor: Peter Schwabe
Second Reader: Joeri de Ruiter

September 28, 2018

Contents

Chapter 1

Introduction

1.1 Introduction

Often when we exchange messages we want to protect our communication from prying eyes. When we communicate over the internet this requires a *cryptographic protocol* since messages may need to pass by many potential attackers before arriving at their destination. Think of eavesdroppers on the wire, but legitimate third parties supposed to pass on messages such as internet service providers are also considered untrusted. We will consider a model in which untrusted parties will, for one, attempt to read the internet traffic between two peers. But we need more, namely in addition to this we want to ensure messages arrive unmodified. Lastly, on the internet we can not visually confirm who we are talking to. Therefore, since the attacker may attempt to impersonate anyone, we require a reliable method for identification.

In the previous paragraph we identified the three major intuitive objectives for a protocol aiming to achieve secure communication over the internet. Recapping, these three major objectives are:

- Confidentiality
- Integrity
- Authentication

To accomplish these goals only limited resources are available. We will assume only a reliable, in-order, data stream. This is provided by TCP.

1.1.1 Transport Layer Security

The requirements identified are met by Transport Layer Security (TLS). This protocol is considered the most important cryptographic protocol in use on the internet today. Given its prevalence it has a fundamental role in internet security. Unfortunately such an important protocol shows its age in numerous ways. TLS is embedded in many small devices around the globe, this makes integrating any updates to TLS extremely difficult. Since breaking backwards compatibility is therefore impossible, many new versions just extend the previous by going yet another layer deeper, TLS 1.2 for example still advertises itself as TLS 1.0 in its `Version` field while using the `client_version` field to indicate the higher version. This also has an effect on implementations, some of which have become increasingly complex throughout TLS' lifetime. The first version of TLS, SSL 1.0, was actually never published. The first publicly released version of what would become TLS was SSL 2.0, developed by Netscape and released in 1995. Since 1999, the protocol has been developed by the Internet Engineering Task Force (IETF). This group published TLS 1.0 in 1999, TLS 1.1 in 2006 and TLS 1.2 in 2008.

TLS actually consists of two separate protocols. First of which is the *handshake protocol* which authenticates one or both of the communicating parties to the other and establishes the parameters and cryptographic keys for the session. This uses for the most part, slower, *public key* cryptography to authenticate and exchange keys. The handshake protocol is designed to withstand tampering by adversaries, who should not be able to influence the cryptographic parameters negotiated by the legitimate parties. Second is the *record protocol* which protects the traffic between the two endpoints with the established cryptographic context. This uses faster *symmetric key* cryptography to ensure confidentiality and integrity of the transmitted data, which is divided in *records*.

Quite recently a new version [1] of TLS, version 1.3, was submitted to the Internet Engineering Steering Group (IESG) for publication. This is the final step before the standardization of TLS 1.3. This new version of the protocol features a major overhaul of the handshake protocol. A major highlight of version 1.3 is that a full handshake now requires just one round trip between client and server. This reduces the time needed to set up the secure channel. Furthermore all public key cryptography now uses (EC)DH[1], moving away from RSA [2] and improving speed in two ways. First by using a faster

[1] Explained in Chapter 2

algorithm and second by reducing the amount of bytes transferred during the handshake. Yet another highlight is the cleanup of symmetric cipher suites; TLS 1.3 allows only for ciphers supporting authenticated encryption. The protocol negotiates an ephemeral key which is used to start encrypting the handshake as early as possible. The ephemeral key is also used to provide forward secrecy, which is now no longer optional. To conclude these highlights; on session resumption the client is allowed to include data in the first flight, enabling what is called a *0-Round Trip* mode. This will reduce latency when client and server have communicated with each other before.

During TLS 1.3 development there was a proposal to change the way in which the server authenticates itself. This proposal was dubbed OPTLS [3] where OPT stand for OPTimized or One-Point-Three. If OPTLS had been standardized as TLS 1.3 it would have featured a different handshake protocol. Forward secrecy is still achieved by an ephemeral Diffie-Hellman [4] key exchange, but the major difference is in how authentication is achieved. While classically the server authenticates using a signature on the handshake transcript, in OPTLS authentication is done based on static Diffie-Hellman. For this the server provides an additional Diffie-Hellman public key, signed by a Certificate Authority. Using this Diffie-Hellman public key a second key is negotiated. Using this second key the server authenticates the handshake. When RSA was still considered this style of handshake could lead to some major performance gains. In addition it has the advantage of having the whole TLS key exchange rely on only one cryptographic primitive, Diffie-Hellman, simplifying implementations. Finally note that eliminating signatures allows plausible deniability of communication — the used authentication key can not be traced to either party — similar to SKEME.[5]

Unfortunately the OPTLS proposal was ultimately not accepted into the TLS 1.3 standard. Instead a more traditional signature-based approach is used for several reasons. Since static Diffie-Hellman certificates are non-standard, there had to be support for a transition mode. In this mode the server would use its signing key to sign a Diffie-Hellman share. However the use of such a *delegated* credential was deemed not well enough researched. In addition this would require the server to store another secret which could be compromised. Having this secret would allow an attacker to impersonate the server for as long as the Diffie-Hellman certificate is valid. A second reason was that the performance gain is not quite as big when compared against ECDSA [6] or EdDSA [7] signatures. Finally; perhaps the proposal was just “too late”. The IETF was afraid that

careful review would delay the standardization of TLS 1.3. However OPTLS is still an interesting approach to eliminate the need of online signatures. Recently a new Internet Draft appeared [8], potentially reviving a form of OPTLS as an extension to TLS 1.3.

1.1.2 Going post-quantum

As mentioned before; the key exchange protocol is based on elliptic curve [9, 10] Diffie-Hellman and digital signatures. The mathematical structure on which these problems are based is assumed to be unbreakable by classical computers. However there are multiple research efforts ongoing to build a *quantum computer* [11]. Such a device, when built, will be able to utilize Shor's algorithm to compromise all the security offered by RSA and Diffie-Hellman. Although this is a significant engineering challenge, building a large quantum computer is believed to be possible within the next twenty years [12]. Notably even connections that are believed to be secure today can, and are, saved to be decrypted in the future. After a quantum computer is built this data can be decrypted to be analyzed by anyone who captured the encrypted data and possesses the resources to build a quantum computer [13].

At the moment NIST has issued a call for proposals [14] to standardize a post-quantum key exchange algorithm. The call includes proposals for public key encryption, *key encapsulation mechanism* [15] (KEM) and digital signatures.

1.2 Motivation

Before stating the objectives of this thesis we will first state the motivation for these.

1.2.1 OPTLS implementation

Since OPTLS was not standardized no implementation was ever finished. Providing such an implementation would be interesting to evaluate the claims of speed increase. Furthermore the recent internet draft which extends TLS 1.3 with an OPTLS-like authentication mechanism could gain some more traction from an implementation with benchmarks.

1.2.2 OPTLS with KEMs

In the OPTLS paper Krawczyk and Wee suggest it would be possible to use a "KEM-like" cryptographic primitive for OPTLS. As mentioned before; NIST currently has a competition running to standardize a post-quantum KEM. If OPTLS can be realized using a KEM this could be a quite interesting way to design a post-quantum version of TLS. Research is needed to figure out how exactly the key exchange looks using KEMs, or else to solve any potential challenges that may surface.

1.2.3 Privacy friendly

We feel that signatures are not the correct cryptographic primitive for server authentication. When the server provides a signature this can convince *everyone* that this server was involved in communication. But in the TLS setup it would be enough to only authenticate the server to the one specific client it is currently communicating with. This is what OPTLS can provide by offering plausible deniability, further motivating the need for an implementation.

1.2.4 Do post-quantum right

Although it may seem like an obvious solution to just drop in whatever post-quantum signature scheme that NIST standardizes, we feel this might be *terrible* idea. Post-quantum signature schemes have very different performance characteristics from the elliptic curve and RSA signature schemes we are used to. Performance and key sizes are different, and some schemes require to keep state to be secure, something which appears impossible to require from every TLS server.

TLS is ubiquitous, being embedded in almost every connected device. This has proven to make the protocol particularly hard to modify and furthermore means versions stick around for a long time. Breaking backwards compatibility is considered unacceptable. Therefore we will likely have just *one chance* to provide a proper version of post-quantum TLS.

1.3 Objectives

In line with the motivation we identity two major objectives for this thesis.

1.3.1 Implementation

By providing an implementation for a classical version of TLS based on OPTLS we aim to make it possible to evaluate performance. We aim to implement the internet draft from [8]; our implementation extends OpenSSL, the most popular TLS library. By providing such an implementation we contribute to research of a version of TLS without signatures and we make it possible to evaluate the performance thereof.

1.3.2 Research

We aim to investigate how practical it is to use KEMs for OPTLS. If any problems arise with building OPTLS using only KEMs we will discuss these and try to provide solutions based on different cryptographic primitives.

1.4 Thesis overview

Chapter 2 We introduce the cryptographic preliminaries required to follow the thesis and give and overview of the currently most used version of TLS: 1.2. We then do the same for TLS 1.3. We show for both version the difference between the full and abbreviated handshakes.

Chapter 3 In this chapter we discuss and describe OPTLS as envisioned by the designers. We note the advantages it has over the final design of TLS 1.3. We then also describe a variant of OPTLS.

Chapter 4 Here we dive into the inner workings of OpenSSL. We then give a detailed description of our implementation of our implementations of the original OPTLS proposal and the Internet Draft.

Chapter 5 We use our implementations to measure performance, and compare these to one another and TLS 1.3. We give accurate cycle count in the different situations and look into the reason behind the numbers we acquire.

Chapter 6 We discuss a small part of the state of post-quantum cryptography and the future of post-quantum TLS. We have a special focus on KEMs and explain the reason for this.

Chapter 7 Finally we conclude the thesis and give directions for future research.

Chapter 2

Preliminaries

This chapter introduces the different variants of the TLS handshake for protocol version 1.2 and 1.3. Furthermore we go further in-depth on the cryptographic construction blocks mentioned in Chapter 1.

2.1 Cryptographic preliminaries

A protocol like TLS can not exist without layers of cryptography. We have already mentioned some of the building blocks required. In this section we explain what they actually do, and what they can accomplish in our context.

2.1.1 Diffie-Hellman

A cryptographic construction we will often use in this thesis is *Diffie-Hellman* or DH for short. The Diffie-Hellman key exchange allows two parties to establish a shared secret without allowing an eavesdropper to obtain this secret. However while each of the parties can be sure it has established an shared secret with *someone*, the primitive itself provides no authentication. Indeed; if Alice and Bob try to establish a shared secret using Diffie-Hellman, there is no inherent guarantee they will not end up having both a shared secret with Eve, who can then read and forward their messages to the other. In order to prevent this we need to build an *Authenticated Key Exchange*, (AKE) [16, 17]. This construction authenticates both parties, although in the context of TLS often it would often be enough to

authenticate only the server towards the client using an *Unilateral Authenticated Key Exchange* (UAKE) [18].

The "most obvious" way to perform a Diffie-Hellman key exchange is over the integers modulo a large prime. For this one takes a large prime modulus p and a generator g of the multiplicative group $\mathbb{Z}/\mathbb{Z}_p$, both g and p can be public. The secret key is an integer x smaller than the group order of g, the public key is g^x. Now two parties will both have a secret and a public key: (x, g^x) and (y, g^y). The idea behind the Diffie-Hellman key exchange is that one party has x and g^y and can compute g^{y^x}, while the other has y and g^x and computes g^{x^y}. Both parties then obtain g^{xy}, while an observer only sees g^x and g^y, from which computing g^{xy} is believed to be hard [19].
Definition 1 (Diffie-Hellman Problem)**.** Given a cyclic group $\mathcal{G}$ and a generator g of $\mathcal{G}$ and two public values g^x, g^y: find g^{xy}.

It is an immediate consequence that if x can be recovered from g^x then the DHP is solved. This is called the *discrete logarithm problem*:
Definition 2 (Discrete Logarithm Problem)**.** Let $\mathcal{G}$ be a cyclic group of order n, and g be a generator for $\mathcal{G}$. Given an element y of $\mathcal{G}$ the *discrete logarithm problem* is to find an integer x such that $g^x = y$.

The main disadvantage of doing a Diffie-Hellman key exchange over the integers is the existence of sub-exponential attacks on the DLP in this group. As a result of these attacks such as *index calculus* and the *number field sieve* the size of the parameters needs to be quite large. In order to make it infeasible to solve the DLP p should be at least 2048 bits [20]. Furthermore there are issues with reusing a single prime for many sessions [20] and "wrong" choices for which the DLP is suddenly easy to solve.

By 1985 the sub-exponential attacks on the DLP were well-known. Miller and Koblitz independently proposed to instead build cryptography on the *elliptic curve discrete logarithm problem*. For this problem no equivalent sub-exponential attacks are known and as such much smaller key sized can be used. The resulting system, *Elliptic Curve Diffie-Hellman*, uses an elliptic curve $\mathcal{E}$ with a base point G instead of the cyclic group of the integers modulo p. For a secret integer x the public key then becomes xG, the rest of the key exchange works the same. The shared secret becomes xyG, however in this thesis we will use the more usual notation g^x for Diffie-Hellman public keys.

The hardness of the ECDLP depends on the underlying curve [21] but, unlike the integers, for ECDH the same curve can be reused

many times without issue. Therefore certain curves have been found which resist all known attacks [22] and these are standardized. The curves we will use for the TLS handshake are limited to five: The NIST curves P-256, P-384 and P-521 [23] which are Weierstrass curves, and the two Edwards curves Curve25519 [24] and Ed448-Goldilocks [25]. These are all specifically selected to allow for short public keys and fast arithmetic.

An interesting property of the Diffie-Hellman key exchange is that it can be done *ephemerally*. Assume two parties have some established key material. Then if the protocol allows for it at any time during normal communication one party can generate a new secret and compute the corresponding Diffie-Hellman share. The initiating party then sends this over to the other, who performs exactly the same procedure. At the end of this simple one round trip the two parties have a new secret which can be mixed into their already established key material. This provides *forward secrecy* [26]; if the previous key were to be compromised, messages after the re-keying will still be protected.

2.1.2 Key encapsulation

The concept behind a *Key Encapsulation Mechanism* [15] (KEM) is to use asymmetric encryption to encrypt a symmetric key. For this public key encryption is used to derive a shared key, the shared key is then used with standard symmetric ciphers to encrypt and authenticate the actual messages. The advantage is that while public key encryption often works only on valid group elements, a KEM includes a *key derivation function* [27] (KDF) to turn these into uniform random cryptographic keys.

Definition 3 (KEM)**.** A KEM = $(Gen, Encap, Decap)$ is defined as a tuple of three algorithms:

- A probabilistic polynomial-time key generation algorithm Gen, which on input 1^λ for $\lambda \in \mathbb{Z}_{\geq 0}$ outputs a random public key/secret key pair (PK,SK). The exact size and structure of these keys depends on the underlying scheme.

  For $\lambda \in \mathbb{Z}_{\geq 0}$ we define the probability spaces

  - $\text{PKSPACE}_\lambda := \{\text{PK} : (\text{PK}, \text{SK}) \xleftarrow{R} \text{Gen}(1^\lambda)\}$
  - $\text{SKSPACE}_\lambda := \{\text{SK} : (\text{PK}, \text{SK}) \xleftarrow{R} \text{Gen}(1^\lambda)\}$

- A probabilistic polynomial-time encryption algorithm $Encap$,

which on input 1^λ for $\lambda \in \mathbb{Z}_{\geq 0}$ and a public key $\texttt{PK} \in \text{PKSPACE}_\lambda$ outputs a key K and a ciphertext c. For a KEM the length of K is a parameter of the system. The ciphertext is a bitstring.

- $Decap$ is a probabilistic polynomial-time encryption algorithm, which on input 1^λ for $\lambda \in \mathbb{Z}_{\geq 0}$ and a secret key $\texttt{SK} \in \text{SKSPACE}_\lambda$ with a ciphertext c outputs either a key K or `invalid`.

For example let G be a group of prime order p with generator g then a simple KEM based on ElGamal [28] would look as follows:

Gen : $\texttt{SK}$: s random from $\mathbb{Z}_q$, $\texttt{PK}$: $h = g^s$

Enc : r random from $\mathbb{Z}_q$, $a = g^r$, $b = h^r$ and $K = \text{KDF}(b)$ using $\texttt{PK}$ h

Dec : From ciphertext a, recover K as $K = a^s$ using $\texttt{SK}$ s

In some ways this seems similar to a Diffie-Hellman key exchange, however in Chapter 6 we will look closer into exactly in which scenarios a Diffie-Hellman key exchange can be replaced by a KEM and we will also see in what scenarios it is not immediately obvious how to do this.

2.1.3 Non-interactive key exchange

A *Non-Interactive Key Exchange* (NIKE) [29], is an important but relatively overlooked cryptographic principle. It allows two parties who already know each others public keys to establish a shared secret *without any interaction*.

The canonical example of a NIKE is in fact the Diffie-Hellman key exchange. We see that when (x, g^x), (y, g^y) are the Diffie-Hellman keys of Alice respectively Bob and they possess each others public key, then both can compute a key based on g^{xy} without any further interaction being required. Unfortunately no interaction also comes with disadvantages: it is not possible to achieve any forward security. But some other security properties are clear: if Alice's private key is compromised this does not effect the security of keys where Alice is not involved. Furthermore the compromise of one shared key should not affect other shared keys.

Definition 4 (NIKE)**.** A NIKE = $(CommonSetup, Keygen, SharedKey)$ is defined as a tuple of three algorithms with an identity space $\mathcal{ID}$ and a shared key space $\mathcal{SHK}$:

- An algorithm $CommonSetup$ which outputs $params$, the system parameters.

- A probabilistic algorithm $Keygen$, which given the parameters and an ID $\in \mathcal{ID}$ outputs a secret key and the corresponding public key: $\texttt{SK}$ and $\texttt{PK}$, we assume $params$ to be included in $\texttt{PK}$.
- An algorithm $SharedKey$, which given an identity $\text{ID}_1 \in \mathcal{ID}$ and a public key $\texttt{PK}_1$ along with another identity ID_2 and a secret key $\texttt{SK}_2$ outputs either a shared key $K \in \mathcal{SHK}$ or $\texttt{invalid}$.

We furthermore require that for any pair of identities ID_1, ID_2 with corresponding key pairs $(\texttt{PK}_1, \texttt{SK}_1), (\texttt{PK}_2, \texttt{SK}_2)$ the $SharedKey$ algorithm satisfies the following constraint:

$$SharedKey(\text{ID}_1, \texttt{PK}_1, \text{ID}_2, \texttt{SK}_2) = SharedKey(\text{ID}_2, \texttt{PK}_2, \text{ID}_1, \texttt{SK}_1)$$

It can be shown that any secure NIKE can be converted into an (IND-CCA) [30] secure public key encryption scheme. So can the ElGamal KEM that we saw in Section 2.1.2 be seen as arising from the Diffie-Hellman NIKE: here (r, g^r) is the ephemeral key of the sender and the receivers public key is used to construct a shared secret g^{sr}.

Unfortunately the reverse claim is not true [29]; in general we cannot take a IND-CCA secure public key encryption scheme (and thus neither a KEM) and use it to construct a NIKE.

2.1.4 Cryptographic hash functions

A *cryptographic hash function* is a fundamental building block in cryptography. These transform an (almost) arbitrary length input into a short fixed length output. This is useful in many situations where it is not practical to handle large amounts of data.

Definition 5 (Cryptographic hash function)**.** A *cryptographic hash function* is an efficiently computable function $h : \{0,1\}^* \to \{0,1\}^n$ such that h is:

- *preimage resistant*; given $Y \in \text{Img}(h)$ it is computationally infeasible to find $X \in \text{Dom}(h)$.
- *second-preimage resistant*; given X and $h(X)$ it is computationally infeasible to find $X' \neq X$ such that $h(X') = h(X)$.
- *collision resistant*; it is computationally infeasible to find $X, Y \in \text{Dom}(h)$ such that $h(X) = h(Y)$.

For example a cryptographic hash function could be used to commit to a certain answer to some question. One publishes the hash of the answer, but not the answer itself, then once one publishes the

answer everyone can verify that this answer must have been the one committed to by the hash.

In TLS we will see that both parties use hash functions to ascertain they have seen the same messages. By hashing the messages in the protocol so far they both obtain a value that, given the properties of a cryptographic hash we have just seen, can only be the result of that exact message flow. However, anyone can compute these values; by themselves they only proof integrity.

2.1.5 Message authentication codes

Once a shared secret with another party is established we would like to be able to use this to authenticate messages. This can be done with a *message authentication code*. In practice these algorithms are faster than using digital signatures. However since both parties possess the shared secret in principle the authenticated message could have been from either. Thus neither of the parties can cryptographically prove to a third party that a message was sent by the other party. Typically a MAC is appended to the data, proving its authenticity.

Definition 6 (MAC)**.** A MAC = $(Gen, Tag, Verif)$ consist of a three tuple of probabilistic polynomial-time algorithms, a message space $\mathcal{M}$, a key space $\mathcal{K}$ and a tag space $\mathcal{T}$:

- Gen, which on input 1^λ for $\lambda \in \mathbb{Z}_{\geq 0}$ outputs a key $K \in \mathcal{K}$
- Tag, which on an input key $K \in \mathcal{K}$ and a message $m \in \mathcal{M}$ outputs a tag $Tag_K(m) = T \in \mathcal{T}$
- Ver takes input a key $K \in \mathcal{K}$, a message $m \in \mathcal{M}$ and a tag $T \in \mathcal{T}$, then $Ver_K(m, T)$ outputs either `accept` or `reject`.

  Furthermore for all λ and all messages m we have:

$$\Pr[K = Gen(1^\lambda) : Ver_K(m, Tag_K(m)) = \texttt{accept}] = 1$$

The last requirement asserts that the verify algorithm always accepts a valid tag. For a MAC to be secure it needs to be *unforgeable*; the adversary is trying to create a MAC on some message that was not seen yet. This should be infeasible even if given access to an oracle which returns valid MACs on every message other than the one required.

2.1.6 HMAC

In Section 2.1.4 we noted that hash functions by themselves do not authenticate their input. However hash functions can be used to construct a MAC, although the naive way of prepending the key has issues which has led to the more sophisticated *HMAC* [31] approach.

Definition 7 (HMAC)**.** Given a message space $\mathcal{M}$, a key space $\mathcal{K}$, a cryptographic hash function h, a message $m \in \mathcal{M}$ and a secret key $k \in \mathcal{K}$ the HMAC of m under K is given by:

$$\text{HMAC}(K, m) = h((K \oplus opad) \| h(K \oplus ipad))$$

Here $\oplus$ denotes the *exclusive or* and $\|$ denotes *concatenation*.

Other MAC constructions exist, but during the TLS handshake historically HMAC are used as the default way of authenticating after a symmetric key has been established. Modern ciphers output a MAC together with the ciphertext per default but in the next subsection we will see the HMAC construction is useful for another purpose.

2.1.7 HKDF

We will often need to derive cryptographic keys from a Diffie-Hellman value resulting from a key exchange. The cryptographic construction which is used by TLS to to this is a *Hash-based Key Derivation Function* (HKDF) [32].

Such a construction is needed because the Diffie-Hellman value is not uniformly random distributed. Thus it should not be used to directly seed some pseudorandom function. Instead this constructing uses two different phases, first of which is *extract* this extracts from the input keying material a fixed length pseudorandom secret, optionally a *salt* (non-secret random value) can also be used, this defaults to a string of zeroes. Second is *expand*, using the pseudorandom secret we can then expand this secret into several cryptographic keys, optionally when expanding an `info` field can be included. This is what we will see is being used in TLS to generate keys for different purposes from one secret.

Both *extract* and *expand* can be implemented using the standard HMAC primitive. For this during the extract phase the salt is used as the key and the shared secret as the input data. During the expand phase the output of this can then be used as the key, while the input data can be whatever is desired. Usually this will be a

concatenation of the handshake hash and an `info` label indicating the purpose of the output key.

2.1.8 Digital signatures

To proof integrity and authenticity of messages we will use *digital signatures*. For this the sender has a key pair consisting of a public key and a private key. The sender can apply a secret function based on the private key on some piece of data and obtain a signature.

The sender can publish their public key freely. Now anyone receiving a message from the sender can use this public key to verify the signature. This proofs that the message was not modified, and that the signature was created by someone who had access to the private key.

Definition 8 (Digital Signatures)**.** A digital signature scheme over a message space $\mathcal{M}$ and a signature space $\mathcal{S}$ consists of a three tuple of probabilistic polynomial-time algorithms $(Gen, Sign, Verif)$ with the following properties:

- Gen, which on input 1^λ for $\lambda \in \mathbb{Z}_{\geq 0}$ outputs a random public key/secret key pair (PK,SK).

  For $\lambda \in \mathbb{Z}_{\geq 0}$ we define the probability spaces

  - PKSPACE$_\lambda$:= $\{\texttt{PK} : (\texttt{PK}, \texttt{SK}) \xleftarrow{R} \text{Gen}(1^\lambda)\}$
  - SKSPACE$_\lambda$:= $\{\texttt{SK} : (\texttt{PK}, \texttt{SK}) \xleftarrow{R} \text{Gen}(1^\lambda)\}$

- $Sign$ takes a secret key $\texttt{SK} \in$ SKPACE$_\lambda$, a public key $\texttt{PK} \in$ PKSPACE$_\lambda$ and a message $m \in \mathcal{M}$; giving the signature $Sign_{\texttt{SK}}(m) = \sigma \in \mathcal{S}$
- $Verif$ takes a public key $\texttt{PK} \in$ PKSPACE$_\lambda$, a message $m \in \mathcal{M}$ and a signature $\sigma \in \mathcal{S}$ it then outputs either `accept` or `reject`.

Furthermore for all λ and all messages m we have:

$$\Pr[(\texttt{SK}, \texttt{PK}) = Gen(1^\lambda) : Verif_{\texttt{PK}}(m, Sign_{\texttt{SK}}(m)) = \texttt{accept}] = 1$$

The security notion of a signature is very similar to that of a MAC. Again we need it to be unforgeable; if the adversary can obtain a signature on a message he has not seen before, the signature scheme is insecure.

Note that contrary to MACs digital signatures have the property of *non-repudiability*. Once someone signs a message anyone can proof

to a third party that this signature *must* have been created by the owner of the private key.

In practice to establish trust we have *Certificate Authorities*, whose keys are included in browsers and operating systems. This establishes a root of trust; the CAs are expected to sign public keys together with identifying information after verifying this information is in fact correct.

2.2 TLS preliminaries

Every version of TLS released by the IESG has its own RFC. So far there have been three standardized versions. These are TLS 1.0 [33], TLS 1.1 [34] and TLS 1.2 [35]. Internally these versions share many similarities, but also some notable differences. Although this chapter is not strictly required to understand OPTLS; we aim to give the necessary background information required to appreciate the major changes TLS 1.3 introduces, and likewise the changes OPTLS variants introduce to the protocol.

2.3 TLS 1.2

The TLS 1.2 handshake comes in two variants: first the *full* handshake, which establishes a new session between server and client and second the *abbreviated* handshake, which allows the parties to resume an earlier session using stored key material.

2.3.1 Full handshake

The protocol flow corresponding to a full TLS 1.2 handshake is shown in Figure 2.1. When using this handshake two full round trips are needed before any application data can be send.

Going through the flow: the `Client-` and `ServerHello` share basic protocol information, such as which ciphers each parties support. The `ServerKeyExchange` is used only when a cipher supporting forward secrecy is selected. If so it includes the servers ephemeral share. Next the `ServerHelloDone` indicates to the client that no further messages are to be expected from the server. If forward secrecy is negotiated then the client then sends a `ClientKeyExchange` including its Diffie-Hellman parameters, otherwise it generates and

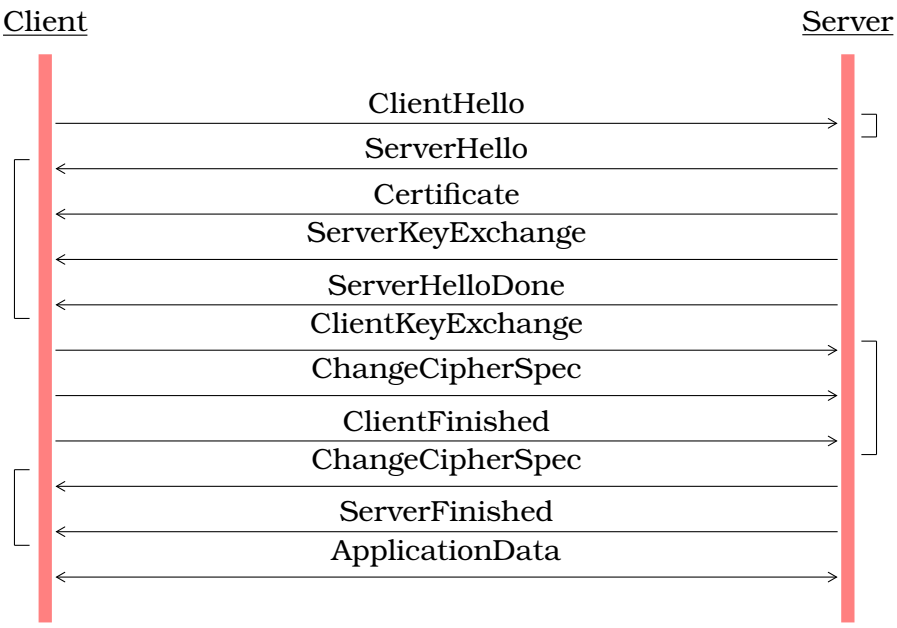

Figure 2.1: Full TLS 1.2 Handshake

encrypts a *pre-master secret* under the public key included in the servers certificate. The `ChangeCipherSpec` record indicates that from now on encryption is used. The final message sent by the client is then `ClientFinished`, which includes a MAC over the whole handshake, allowing the server to verify that both parties saw the same messages. The server ends the handshake by doing the same in its `ChangeCipherSpec` and `ServerFinished`, completing the handshake. After validating the `ClientFinished` the server can send data, the client is only allowed to process this data or send data of its own after validating the `ServerFinished`.

2.3.2 Abbreviated handshake

When server and client have an established *session ticket* they can instead resume this session by including some information identifying the session in the `ClientHello`. The server then indicates if the session is valid by including some information in its `ServerHello`. This leads to the protocol flow seen in figure 2.2. The client can now include data after sending its `ClientFinished`, effectively removing one of the round trips needed before application data can be send. However to retain forward secrecy the session tickets should be updated periodically.

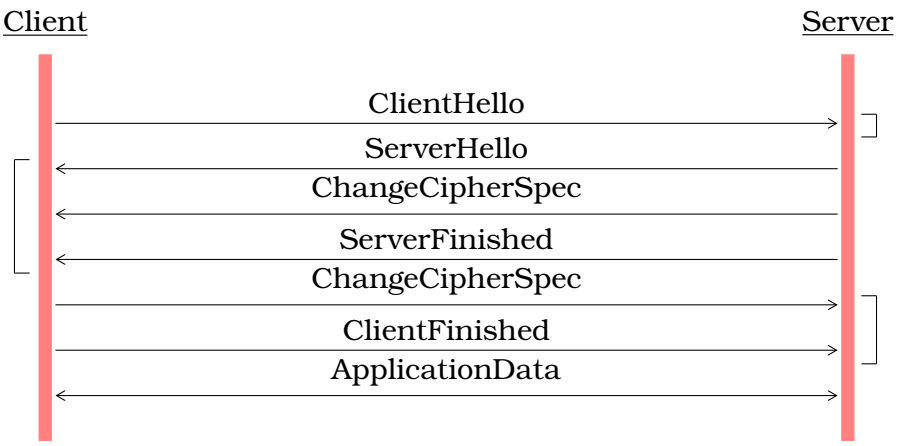

Figure 2.2: Abbreviated TLS 1.2 Handshake

2.4 TLS 1.3

The TLS 1.3 handshake also comes in the same two flavors as the TLS 1.2 handshake. There is a major difference though, in the 1.3 abbreviated handshake the client is allowed to include *early data* in the first flight. This data has some slightly different security properties on which we will expand in Section 2.4.3. Furthermore to lock out passive eavesdroppers early, encryption starts as soon as possible: in the full handshake the certificate is already encrypted.

2.4.1 Full handshake

In TLS 1.3 the full handshake (Figure 2.3) has been changed to include key material already in the `ClientHello` and `ServerHello` records. These both include an ECDH public key. This introduces mandatory forward secrecy. `ChangeCipherSpec` is only included for backwards compatibility; it makes the handshake look similar enough to the TLS 1.2 abbreviated handshake to pass through certain middleboxes [36]. The `Certificate` includes a signing key which is used to sign the messages so far. This signature is then sent in the CertificateVerify. This proofs the server possesses the private key associated with the certificate to the client, and thus — when the certificate is properly signed by some trusted CA — it authenticates the server to the client. Both the `ServerFinished` and `ClientFinished` then assure each party has seen the same messages and finish the authentication process.

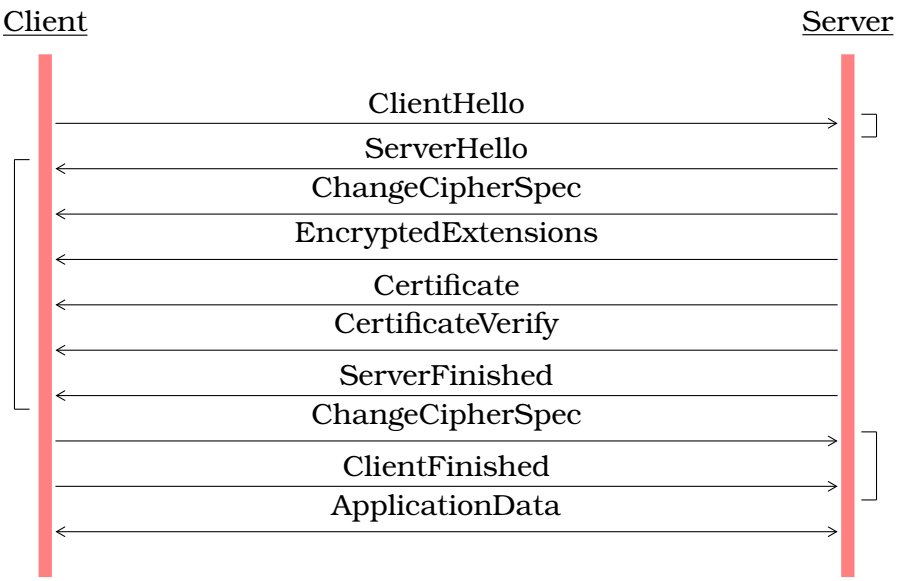

Figure 2.3: Full TLS 1.3 Handshake

2.4.2 Abbreviated handshake

The abbreviated handshake is shown in Figure 2.4. The client can initiate an abbreviated handshake by including a Pre-Shared Key (PSK) in the `ClientHello`. The server indicates in the `ServerHello` if it accepts this PSK. If so the authentication is inherited from the earlier session. An ephemeral Diffie-Hellman exchange is still performed to achieve forward secrecy.

2.4.3 0-RTT

A major feature of TLS 1.3 is that a client is allowed to include early data in the first flight to the server. To accomplish this, previously established key material is used in an abbreviated handshake. This is shown in Figure 2.5. To initiate early data the client includes the `early_data` extension in its `ClientHello`. If the server intends to process this data it replies with its own `early_data` extension in its `EncryptedExtensions`. If the server does not include this extension the client knows the server did not accept the early data. Since the client does not yet have an ephemeral Diffie-Hellman share from the server this data is only encrypted with keys derived from the PSK. Thus the data is not forward secret. Furthermore since the client does not have any server randomness yet the freshness of commu-

nication is not guaranteed: the data could be a replay of previous communication.

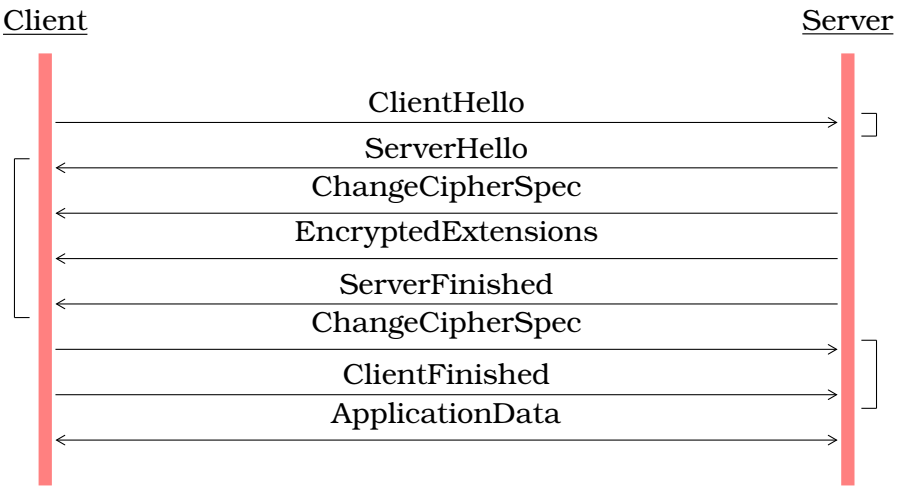

Figure 2.4: Abbreviated TLS 1.3 Handshake

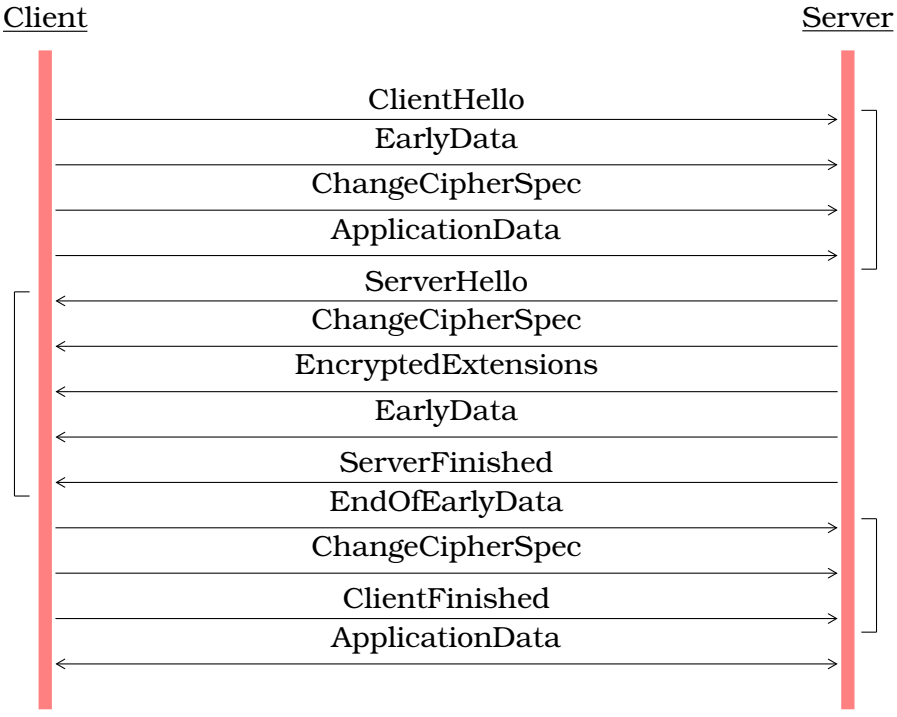

Figure 2.5: Abbreviated TLS 1.3 Handshake with Early Data

Chapter 3

OPTLS

The simplest way to describe OPTLS is "TLS without signatures". The original paper on OPTLS by Krawczyk and Wee [3] describes the protocol as a full-fledged new version of TLS. Such a new version would have offered considerable freedom to optimize TLS for the future. This did not happen and TLS 1.3 draft-09 [37] was the last revision of this style before ultimately a more conservative approach was decided on for the new standard. In March 2018 an Internet Draft by Rescorla and Sullivan [8] revived the basic idea as an extension to TLS 1.3. In this chapter we first give a high level overview of OPTLS. We then describe the protocol flow and key derivation schemes for both versions. Details on our implementation of both into OpenSSL will follow in Chapter 4.

3.1 OPTLS

OPTLS is at its core a simple one-round-trip protocol between a server and a client depicted in Figure 3.1. Both the client and server provide unique session nonces: n_C and n_S. They also provide negot$_C$ and negot$_S$, which encompass protocol parameters such as version, available cipher suites, etc. For the rest of the chapter the client has an ephemeral Diffie-Hellman key g^x while the server is assumed to be in possession of a CA-signed Diffie-Hellman certificate cert$_S$. This certificate binds the server's identity to a Diffie-Hellman public key g^s, the fundamental difference s thus that the server's identity is not bound to a signing key. Instead the server can authenticate itself to the client by providing a MAC over the handshake transcript with a symmetric key derived from the shared secret g^{xs}. Furthermore we

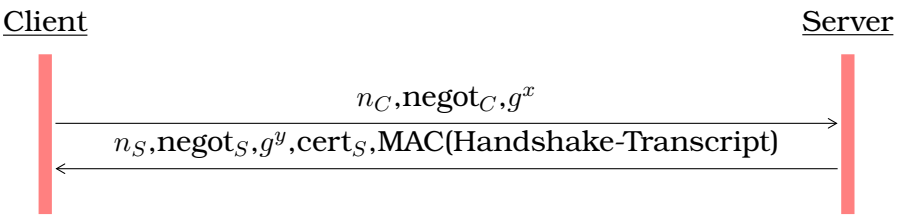

Figure 3.1: Core OPTLS flow

also assume the server has an ephemeral Diffie-Hellman key g^y, this can be used to establish perfect forward secrecy: the session key is ultimately derived from both g^{xy} and g^{xs}. This provides confidentiality of past communication case the server ever leaks its private key s.

3.1.1 On the transition from signing certificates

We have seen that in TLS 1.2 and 1.3 the servers certificate always contains a signature public key. Thus, unsurprisingly, historically servers do not typically possess Diffie-Hellman certificates. In fact Diffie-Hellman certificates are rarely supported by CAs. To accommodate the transition to Diffie-Hellman certificates the OPTLS protocol allowed the server to create a *delegated credential*. The idea was that this would allow servers to use their traditional signing key to validate a static Diffie-Hellman public key g^s. Thus a server sends its (signing) certificate to the client and uses the corresponding private signing key to either sign: (i) g^s with some included validity period or (ii) g^s with the client's nonce n_C. Both of these options are intended to limit the validity of g^s. Option (i) has the disadvantage of allowing an attacker who somehow obtains the server's private key to create credentials which can be used to impersonate the server for the validity duration. Thus the validity duration should be limited. However unlike option (ii) this does not require online signatures, which can be costly in terms of performance and reintroduce signatures into the handshake. An advantage of option (ii) is that the validity of g^s is limited to one session.

3.1.2 0-RTT

As required by the TLS 1.3 working group, OPTLS allows the client to include data in the first flight to the server if key material has been

established during previous communication. To encrypt and authenticate this data a key derived from g^{xs} is used. However, like TLS 1.3, OPTLS does not include protection against replays of `early_data` at the protocol level. To protect against replays a server could keep a state and reject duplicate messages or could require `early_data` messages to be *idempotent*, meaning they should not contain information that might change state. For example, when TLS is used for HTTPS this could mean accepting only `GET` requests without parameters. This is not entirely trivial to solve at the transport level and furthermore the problem is not specific to OPTLS, but instead to the protection of `early_data` in general. For more details on how the attack looks we refer to [38, Sec. 7.1.2].

3.1.3 Plausible deniability

Removing signatures from TLS according to the OPTLS proposal has the advantage that this enhances privacy features of the protocol. By eliminating signatures the server gains *plausible deniability* of the client's identity. Indeed the signature on the handshake transcript in TLS 1.3 can be verified by anyone with the servers public key. Thus the server irrevocably confirms to everyone that it has communicated with someone, an observer may be able to link this with a `ClientHello`. If the observer knows who sent this `ClientHello` he has proof of communication between two parties. In contrast, with OPTLS the MAC can only be verified with the key obtained by the key derived from g^{xs}, which is known only to the participants of a handshake.

3.1.4 Key derivation

OPTLS uses a tree-like structure for key derivation based on the standard construction of HKDF-Expand and -Extract [39]. The diagram for key derivation is shown in Figure 3.2. Input to the respective HKDF functions is based on exactly which type of handshake is performed. We study the three main modes.

1. Session resumption based on a pre-shared key (PSK) or resumption secret.
2. Similar to the previous, with an ephemeral Diffie-Hellman key exchange.
3. The "full handshake": 1-RTT semi-static.

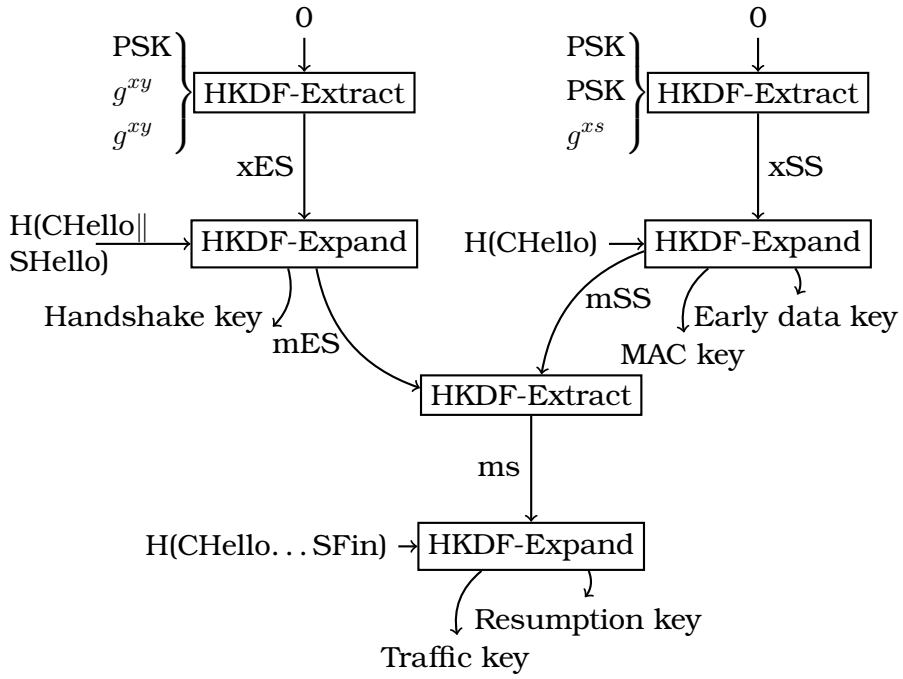

Figure 3.2: OPTLS Key Derivation Tree

The input to the HKDF-Extract functions in the diagram are the same order. We can that see on the left side of Figure 3.2 the Ephemeral[1] Secret is used to generate a key which encrypts the rest of the handshake. On the right side a Semi-Static secret is used to derive keys for the MAC and the 0-RTT data. Input to the HKDF-Extract functions also includes a hash of the message transcript seen so far. Notably: on the right side this does not yet include the `ServerHello`, as the client has not seen this yet when sending early data. Both sides of the tree then come together and are used to generate the master secret, from which the keys are derived to encrypt `application_data`. We will return to the exact implementation in Chapter 4.

3.1.5 State machine

The main difference with the TLS 1.3 state machine is that in OPTLS there is no need to send a `CertificateVerify` message [40]. Thus in the implementation we will need to add support to go right from the `Certificate` message to the `ServerFinished`. Unfortunately skipping

[1] Here 'E' stands for "Ephemeral and not for "Early".

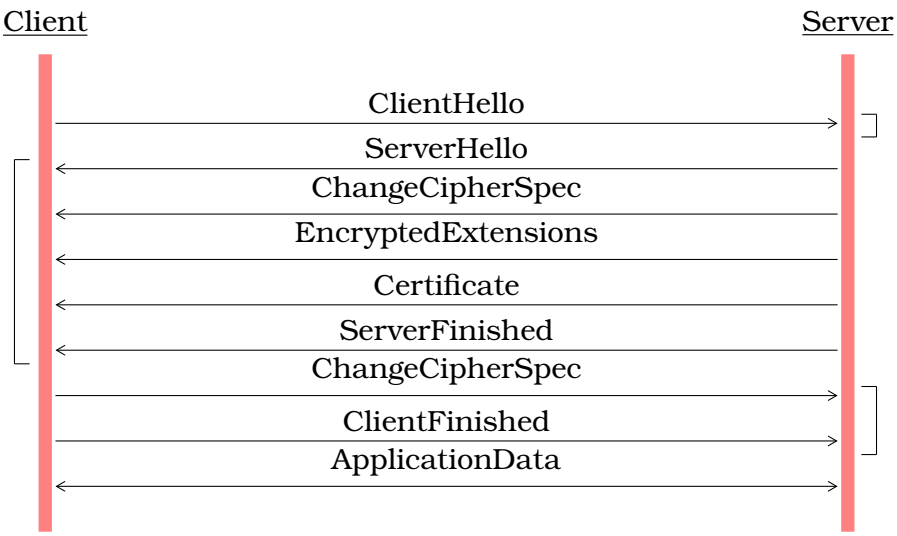

Figure 3.3: Full OPTLS Handshake

messages like this will likely confuse some internet infrastructure, and we will see that the Internet Draft retains the `CertificateVerify` message. The message flow of a full handshake is shown in Figure 3.3. The rest of the state machine does not need to change.

3.2 Internet-Draft: TLS 1.3 Semi-Static KX

Now that we have an overview of OPTLS we will see how the adaptation into TLS 1.3 might look. Note that this Internet Draft is still in an early stage — at the moment of writing TLS 1.3 is still in final stages of standardization. Therefore perhaps interest in extensions to TLS 1.3 may also be in an early stage. It is quite possible for the Internet Draft to significantly change, nonetheless this appears to be the main avenue to an OPTLS-based standard.

3.2.1 Negotiation

To negotiate the OPTLS-like extension to TLS 1.3 we advertise the MAC as if it were a signature by adding several new valid "signature" algorithms. These will indicate that a MAC is used. If the client supports these, then the client is is allowed to include one or more of these signature algorithms in the `signature_algorithms` of

the `ClientHello`. If the server selects one of these algorithms then it will send a certificate containing a Diffie-Hellman share g^s.

3.2.2 Cryptographic keys

The static secret g^{xs} is used in two places to derive keys. First the MAC key for `CertificateVerify` is derived from g^{xs} by one HKDF-Extract: HKDF-Extract(0, g^{xs}). This is called the SS-Base-Key and from this the actual key is derived using HKDF-Expand as in TLS 1.3, it seems likely the label (currently "finished") will change during development of the RFC, however this only requires a minimal change in the implementation.

Second the static secret is mixed into the TLS 1.3 key derivation tree in the last stage, where during the normal TLS 1.3 handshake only zeroes are mixed in. The modification is shown in Figure 3.4, which is heavily based on the relevant portions of the TLS 1.3 RFC section 7.1 and shows where g^{xs} is input in red.

3.2.3 Protocol flow

A major advantage of this approach is that there is no need to change the state machine of TLS. This furthermore makes sure we retain backwards compatibility to all previous versions of TLS, as is to be expected of an extension of TLS 1.3.

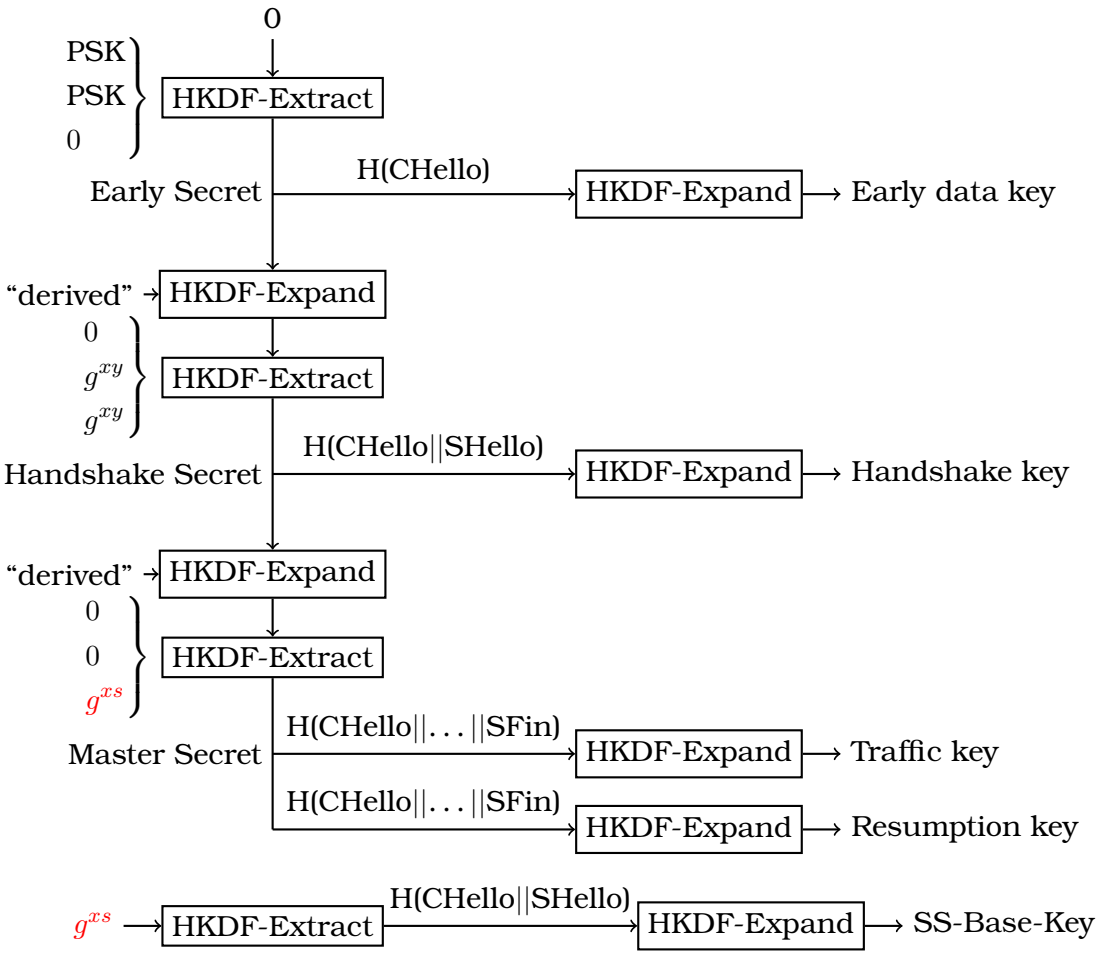

Figure 3.4: Internet Draft Key Derivation Tree, from the TLS 1.3 tree

Chapter 4

Implementation

We implement the two styles of OPTLS, i.e, the original proposal and the Internet Draft into `libssl` of the ubiquitous OpenSSL[1] library. OpenSSL is not commonly known as an easy-to-modify piece of software. In fact the man pages of `s_client` and `s_server` contain the following warning in the `BUGS` section:

> "Because this program has a lot of options and also because some of the techniques used are rather old, the C source of `s_client` is rather hard to read and not a model of how things should be done. A typical SSL client program would be much simpler."

Thus although this chapter will focus on our implementations, we further hope to give a detailed enough overview of the inner workings of OpenSSL that future work can build on our efforts.

4.1 Hacking on OpenSSL

First off, we recommend adding compilation flags to enable `gdb` to access constants and structures defined in the source code. This can be done by adding `-ggdb3 -g3` to the `CFLAGS` in `10-main.conf`. This will definitely prove to be useful when stepping through the code; do not forget to pass `--debug` to `./configure`!

Here follows the global overview of relevant OpenSSL functionality:

[1] http://www.openssl.org

- The two programs we will modify are `s_client` and `s_server`. Both of these have C files named after them. Their main functions are `s_client_main` and `s_server_main`. These functions allocate memory for and, where necessary, initialize SSL/TLS structures, parse the command line options and then connect using the specified protocol or listen for a connection.
- All the defines for SSL/TLS structures are in `ssl_locl.h` with `typedefs` for these structures in `ossl_type.h`. It can be challenging to figure out exactly why or where a certain field is in one of these structures. Some important fields: on a valid session resumption `hit` is set in the encapsulating `SSL` structure. This structure also contains all the secrets. It contains a structure `SSL_METHOD` which again holds a structure `SSL3_ENC_METHOD`; here are stored important callback functions for cryptographic operations. For example the `change_cipher_state` field, which is initialized with a function that changes the cipher state based on the state of the connection. Another example is the `tmp` structure, which contains several connection specific parameters, such as the selected certificate and the signature algorithms supported by peer.
- The most interesting functions of the TLS state machine are implemented in `statem_clnt.c`, `statem_srv.c` and `statem_lib.c`. In the first two files the state machine transitions are defined. The functions are all named `ossl_statem_*`. Especially interesting for the state transitions of TLS 1.3 are four functions, these are:
  - `ossl_statem_client13_read_transition`
  - `ossl_statem_client13_write_transition`

  for the client, and their analogues for the server. The records are handled by functions such as `tls_construct_client_hello` or `tls_process_server_hello` for sending and receiving respectively. `statem_lib.c` is similar; it provides functions for the records that both client and server need to handle; for example `tls_process_finished`. Lastly: `statem.c` mostly contains general functionality for switching between reading and writing mode, which is not so interesting for our purposes.
- Each extension has an initializer, client and server side parsers, constructors and a finalizer. These functions can be found per extension in the `ext_defs` array in `extensions.c`. The initializers and finalizers are also implemented in this file. Parsers and constructors for server and client are in `extensions_clnt.c`

and `extensions_srv.c`. These functions read or write the actual bytes to the packet structure.

- The encryption of TLS 1.3 is implemented in `tls13_enc.c`. Here functions such as `tls13_derive_key` — which derives the actual cryptographic key from a secret — are located. Another important function implemented here is `tls13_change_cipher_state` which can perform several actions such as update the hash transcript or derive secrets, based on the handshake state. Note that the actual implementation of the ciphers and other primitives is in `libcrypto`.

4.1.1 Implementing OPTLS

Here we describe our implementation based on TLS 1.3 draft-09 and the OPTLS paper.

Negotiation

To support TLS 1.3 in many places OpenSSL uses the `SSL_IS_TLS13` macro to influence control flow, this macro checks if the protocol version is TLS 1.3 or bigger. Since we will negotiate a higher version we will still hit all these statements, but we need to change the flow in some places. Thus the first thing we need is our own `SSL_IS_OPTLS` macro, which checks the relevant field of the `SSL_CTX` structure for the version.

Since the protocol flow is still mostly identical to TLS 1.3 we first look for all the places where `SSL_IS_TLS13` is used. We add checks for these code paths to also be taken (or not) by adding `SSL_IS_OPTLS` control flow statements.

To then include the actual bytes for the new version of TLS we follow the same procedure as TLS 1.3, we set the `ClientHello` version to 1.2 and rely on the `supported_version` extension to indicate to a server that we support OPTLS. For this we need to modify one of the extension construction functions, in this case `tls_construct_client_hello`.

We further change the client state machine to not accept client certificate requests if we negotiated OPTLS. After a `Certificate`, the next message the client receives should be `Finished`, this is also changed in `statem_clnt.c`.

The server state machine is modified too, first off the server never sends a certificate request so we remove this state machine transition if we negotiated OPTLS. Second after sending the `Certificate` the next state is sending the `Finished`.

Cryptographic Operations

To add the cryptographic operations we require we first need to add support for Elliptic Curve Diffie-Hellman certificates so that we can obtain the keys contained therein. Adding new certificates can be done quite easily since OpenSSL is able to parse the DH certificates without any issues; they are just not used for TLS. We add support by defining a new constant for the certificate type in `ssl_locl.h`. Internally OpenSSL keeps an array of these certificates, the constant is then the index where this certificate type is stored; meaning that the order matters. We place our new certificates at the end and also add a constant which indicates where DH keys start.

In `tls_choose_sigalg` the server[2] chooses which certificate to use based on the available certificates and the `signature_algorithms` extension. When using OPTLS we change this to just check for the presence of a DH certificate. It is an error to negotiate OPTLS without an available DH certificate.

Recall that the server has the static public key g^s and the client the ephemeral public key g^x. The new shared secret g^{xs} needs to be derived for both server and client. For the server this happens in `ssl_derive` in `s3_lib.c`. If this function is called and we are not resuming a session then we generate g^{xs} and derive the static secret with the private part of the DH certificate and the clients ephemeral public key. The ephemeral secret is also derived in `ssl_derive` for both client and server. Both secrets are saved as part of the context.

The client can construct the static secret only after parsing the certificate. This happens in `tls_process_server_certificate`. After extracting the servers public key the shared secret is derived, it can then immediately be used to derive the static secret.

[2] And the client, but we do not consider client certificates

Session resumption

We support two ways to resume a session. The main difference is the value that is used to derive the ephemeral secret. This is decided when the `key_share` extension is processed[3]. For PSK-mode the client has determined in the finalizer of the `key_share` extension that it has not actually sent the extension. Thus, if we actually are resuming, we reach the point in the finalizer (`final_key_share`) where we can HKDF-Extract the ephemeral secret with 0 and the PSK as input. On the servers side it determines during `key_share` construction in `tls_construct_stoc_key_share` that it did not parse a `key_share` from the client, unlike the client the server then immediately derives the same secret.

Alternatively we have PSK-DHE mode, for both server and client the ephemeral exchange is done in `ssl_derive`. It requires no further changes to the control flow from TLS 1.3.

We derive the master secret from both the static and ephemeral secret. To achieve this we introduce `optls_generate_master_secret`. This function expands the ephemeral secret before its fed into HKDF-extract with the static secret. This enables us to construct the "tree-like" structure of OPTLS key derivation. This is done in `optls_enc.c`.

Lastly we add support for the `-optls` and `-no_optls` command line options, to en- or disable OPTLS. They can be used with both `s_client` and `s_server` and influence which TLS version is included in the `supported_versions` extension in the expected way.

4.1.2 Implementing the Internet Draft

Since this incarnation of OPTLS is specifically designed to be integrated into TLS 1.3 it should not come as a surprise that this leads to a cleaner implementation. This is an implementation of the Internet Draft.

Negotiation

For negotiation we need to add several signature algorithms. These need to be added to the `sigalg_lookup_tbl` in `t1_lib.c`. The lookup table includes a field for the type of certificate that is required for

[3]The initializer and finalizer for an extension are still called when the extension is not present

each algorithm. Here we enter the new values of our Diffie-Hellman certificates. The client will offer these in its `supported_sigalgs` extension. When the server looks up the certificate needed for the new `sigalg` values it correctly selects one the DH certificates or tries to fall back to another `sigalg` if these are unavailable. Note that the term `sigalg` is overloaded here: usually it refers to which signature algorithm the server uses, but here it instead indicates that there will not be any signature.

Cryptographic Operations

If we have selected a Diffie-Hellman certificate then we need to modify the authentication mechanism. We can use this selection as a criteria for control flow as these certificates have known indices; we check if the index is larger than `SSL_PKEY_DH_CERT_START`.

For the server we derive g^{xs} again in `ssl_derive`. For the client however we wait until we process the `CertificateVerify`, because from the client's perspective there we first deviate from TLS 1.3; we do not need g^{xs} anywhere earlier and in this location we can cleanly branch between the two forms of authentication.

Starting with the server: we change `tls_construct_cert_verify` to compute a MAC over the handshake so far if a DH certificate was selected. First we generate the SS-Base-Key from g^{xs}, then we compute the MAC over the hash of the handshake transcript.

For the client we do something similar: in `tls_process_cert_verify` we again branch between authentication styles based on if a DH certificate was selected or not. As mentioned before; the client then derives g^{xs} and uses it to compute the SS-Base-Key, after which it can verify the first MAC.

The code for the second MAC in the `ServerFinished` does not need to change. The only thing that further needs to be modified is the input for the HKDF-Extract of the Master Secret. This is done by checking if a DH certificate was negotiated in `tls13_generate_secret` and selecting the input based on that.

Chapter 5

Benchmarks

We benchmark the performance of our implementations using the timing tool provided by OpenSSL: `s_time`. This tool tries to establish as many connections as possible in a given time frame and reports back how many connections it was able to establish per second. However some challenges need to be overcome before this tool is fit for all our purposes.

First of we add support for session resumption to `s_time`. For these benchmarks we hard-code a PSK which can be used to test the performance of PSK-DHE and PSK modes.

Then secondly `s_time` lacks the `-curves` option which controls the elliptic curve selected for the client ephemeral Diffie-Hellman key. The easiest way to solve this, is to change the default curve, we change the order in `t1_lib.c`: where they are in order in the array `eccurves_default`. This allows us to also evaluate performance of X448 and the NIST curves.

Thirdly there is no option for `s_client` or `s_server` to disable PSK_DHE mode. The way we get OpenSSL to negotiate PSK mode without ephemeral DH is to remove the `TLSEXT_KEX_MODE_FLAG_KE_DHE` option. We need to disable this constant in `tls_construct_ctos_psk_kex_modes`. If the server is then also configured to allow PSK only mode we can measure performance.

We disable certificate chain verification since there is no difference between OPTLS or TLS 1.3. This would only introduce a constant overhead in our measurements.

5.1 Results

We start by measuring the total amount of connections possible with both server and client running locally. This gives a first impression of the performance gains of OPTLS. However for TLS it is often more significant how the server performs since this side needs to handle significantly more connections on one machine. Therefor we continue this section by continuing to look closer into the performance of client and server separately when using OPTLS and compare it against TLS 1.3.

5.1.1 Connections per second

- We set a baseline by measuring TLS 1.3 performance. The main thing we are interested in is how the performance differs when performing a full handshake using a signing key versus a Diffie-Hellman key. To that end we measure here using two different signing keys. First an RSA key, which we expect to be slower than the second key, which is an Ed25519 key. We also measure performance when using PSK-DHE and PSK only. The ephemeral key exchange algorithm used is X25519.
- Next we measure OPTLS. We look at the full handshake using X25519 for Diffie-Hellman. We also again look at PSK-DHE and PSK.
- For the Internet Draft we implement all the suggested Diffie-Hellman algorithms, that is: X25519, X448, P-256, P-384 and P-521. We measure the performance of all of these. Since there is no change for PSK-DHE and PSK from TLS 1.3 we do not measure these again.

We perform 50 measurements of 30 seconds each. All measurements are performed on an Intel Haswell i7-4770K CPU 3.50GHz processor. Our implementations are both based on `OpenSSL version 1.1.1-pre10-dev`, commit `7d38ca3f8b`. All measurements are on a single core with Turbo Boost and hyper-threading disabled. We configure the used cipher to be `TLS_CHACHA20_POLY1305_SHA256` in every case and all certificates have an Ed25519 signature. We use gcc version 6.3.0 and the full configure command is:

```
./config enable-ssl-trace enable-ec_nistp_64_gcc_128 -DMYBENCH
```

The results of our measurements are shown in Figure 5.1.

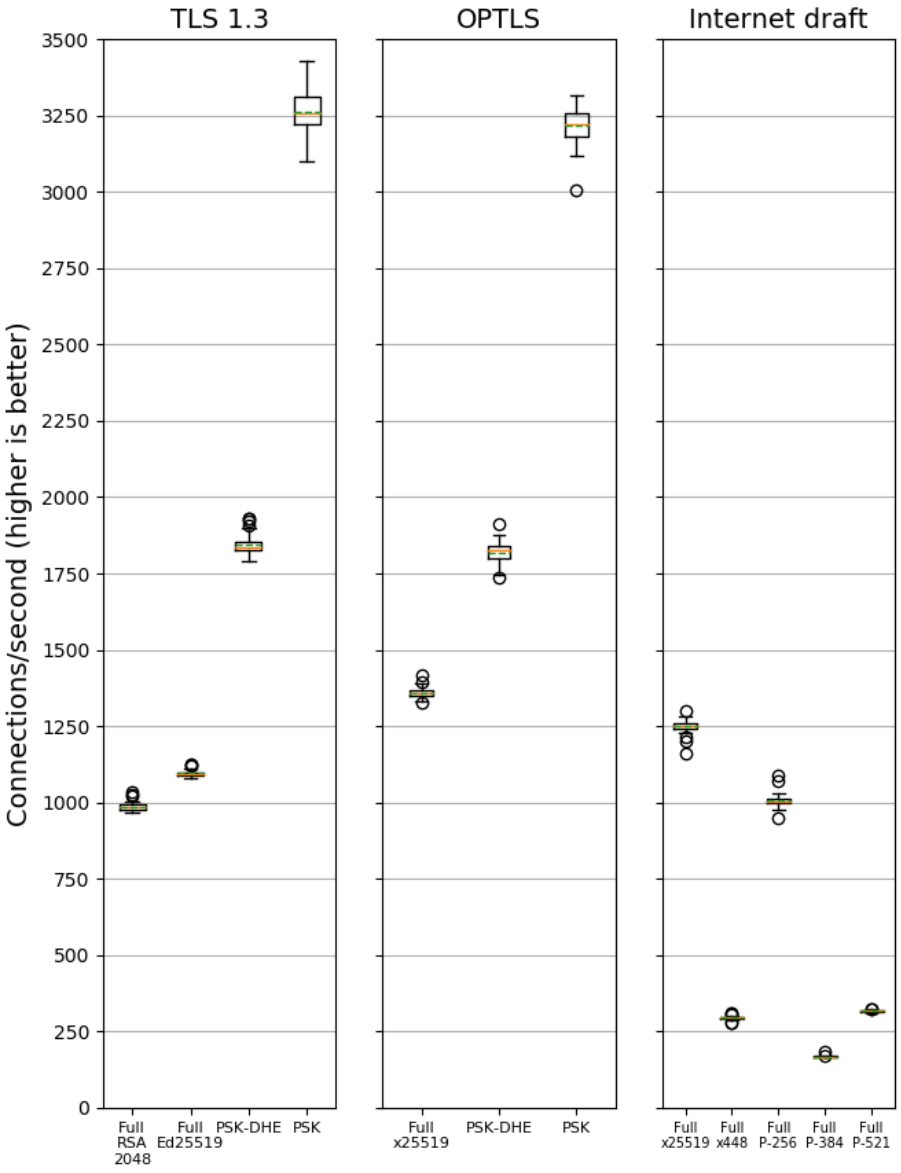

Figure 5.1: Performance Measurements

5.1.2 Server performance

We measure server performance by counting the total amount of clock cycles spent inside several of the cryptographic primitives. This is to reduce the noise that may be introduced by waiting for data to arrive. Also considering that we are interested in post-quantum research we note that we measure some parts which are identical between TLS and OPTLS, but for which different algorithms must be chosen to achieve post-quantum security in the future.

More specifically we measure the cycles used by the server to:

- Generate the ephemeral key share included in the `ServerHello`.
- Derive the early secret by computing g^{xy}.
- In TLS 1.3: Compute the signature over the handshake transcript.
- In OPTLS and the Internet Draft: Derive the static secret g^{xs}.
- For the Internet Draft: compute the first MAC, which is included in the `CertificateVerify`.

In practice this means that we count the total cycles used by the functions `ssl_generate_pkey`, `ssl_derive`, `tls_construct_cert_verify`.

To obtain accurate measurements we use a small snippet of inline assembly which is included when we configure with `-DMYBENCH`. We initialize the cycle counter to 0 and use a subtraction and addition pattern to obtain accurate cycle counts in between the measurements.

The average number of clock cycles used by the server across 1000 connections is shown in Table 5.1 for TLS 1.3, here signatures are used for authentication, the last column shows bytes written and read by the server for the full handshake.

In Table 5.2 we show the performance of the handshake when using our implementation of the internet draft. Of course here we are required to match the curves used for the ephemeral key exchange and the `sigalg`.

And finally, in Table 5.3, we give the benchmark of OPTLS.

Table 5.1: TLS 1.3 Server Benchmarks

Ephemeral	Sigalg	Avg. Cycles	w/r
X25519	ed25519	451438	283/791
P-256	ed25519	495493	283/791
X25519	ecdsa_secp256r1_sha256	469525	316/840
P-256	ecdsa_secp256r1_sha256	490891	316/840
X25519	ecdsa_secp384r1_sha384	3814498	283/733
P-256	ecdsa_secp384r1_sha384	3840982	283/733
X25519	ecdsa_secp521r1_sha512	1453955	316/782
P-256	ecdsa_secp521r1_sha512	1483818	316/782
X25519	rsa_pss_rsae_sha256 (2048)	2467698	283/792
P-256	rsa_pss_rsae_sha256 (2048)	2500159	283/792
X25519	rsa_pss_rsae_sha256 (4096)	22636946	316/842
P-256	rsa_pss_rsae_sha256 (4096)	22654267	316/842

Table 5.2: Internet Draft Server Benchmarks

Ephemeral	Sigalg	Avg. Cycles	w/r
X25519	sig_x25519	522123	269/645
X448	sig_x448	3616417	293/710
P-256	sig_p256	639145	302/742
P-384	sig_p384	9938938	334/803
P-521	sig_p521	3535897	370/877

Table 5.3: OPTLS Server Benchmarks

Curve	Avg. Cycles	w/r
X25519	486036	285/583
X448	3575601	309/648
P-256	595333	318/680
P-384	9839238	350/741
P-521	3502475	386/815

5.1.3 Client performance

We are also interested in the client's performance. We measure how many cycles the client uses to:

- Generate its ephemeral key share.
- Like the server: compute the early secret g^{xy} and for the OPTLS variants also g^{xs}.
- Verify the MAC in the `CertificateVerify` and in TLS 1.3 also the signature.

We use the same method as for the server, the functions that we benchmark are the analogues of the ones on the server side, namely: `ssl_generate_pkey_group`, `ssl_derive`, `tls_process_cert_verify`. For OPTLS we also benchmark `tls_process_server_certificate` partially: for the client g^{xs} is derived here.

The results of the baseline benchmark for TLS 1.3 are shown in Table 5.4.

Table 5.4: TLS 1.3 Client Benchmarks

Ephemeral	Sigalg	Avg. Cycles	w/r
X25519	ed25519	752235	283/791
P-256	ed25519	803925	283/791
X25519	ecdsa_secp256r1_sha256	711670	316/840
P-256	ecdsa_secp256r1_sha256	739876	316/840
X25519	ecdsa_secp384r1_sha384	3093090	283/733
P-256	ecdsa_secp384r1_sha384	3137300	283/733
X25519	ecdsa_secp521r1_sha512	2598950	316/782
P-256	ecdsa_secp521r1_sha512	2636928	316/782
X25519	rsa_pss_rsae_sha256 (2048)	520727	283/792
P-256	rsa_pss_rsae_sha256 (2048)	565813	283/792
X25519	rsa_pss_rsae_sha256 (4096)	796483	316/842
P-256	rsa_pss_rsae_sha256 (4096)	840755	316/842

And for the client Table 5.5 and Table 5.6 show the results for the two respective variants of OPTLS.

5.2 Discussion

Analyzing the results we see that Diffie-Hellman based authentication is faster than RSA signature based authentication as expected

Table 5.5: Internet Draft Client Benchmarks

Ephemeral	Sigalg	Avg. Cycles	w/r
X25519	sig_x25519	554960	269/645
X448	sig_x448	3651390	293/710
P-256	sig_p256	693379	302/742
P-384	sig_p384	9970935	334/803
P-521	sig_p521	3580660	370/877

Table 5.6: OPTLS Client Benchmarks

Curve	Avg. Cycles	w/r
X25519	501263	285/583
X448	3601070	309/648
P-256	638396	318/680
P-384	9954724	350/741
P-521	3543938	386/815

when looking at the performance of client and server combined. Interestingly enough we even see a significant increase in the amount of connections established using OPTLS when compared to the faster Ed25519 signature algorithm.

We measured the cycle counts for an Ed25519 signature and for an X25519 key derivation; indeed as expected Ed25519 requires less cycles than X25519. The former needs about 125000, while the latter needs up to 167000. This is in line with what is expected theoretically. In Ed25519 the costly operation is a fixed base point multiplication, for which precomputation tables can be built. This is not possible for X25519 since here the point is variable; it is supplied by the connecting client.

Our OPTLS implementation requires less cycles on average for cryptographic computations than our implementation of the Internet Draft, this is expected; the former does not send a `CertificateVerify`, while the latter does. The performance difference can be explained mainly by the time it takes to construct the extra `CertificateVerify` record and compute the MAC included in it.

On the server side the Internet Draft needs about 40000 cycles more than OPTLS. We have looked closer at the benchmarked functions to explain where the difference comes from, it can be explained as follows:

- The Internet Draft requires 11000 cycles more for `ssl_derive`.

This is because an extra HKDF-Expand is required when compared to OPTLS. For OPTLS on both sides of the tree we have an input of zeroes, so no pre-extract derive secret is required. In the Internet Draft this is required for the handshake secret, so there is one additional HKDF-Expand when compared to OPTLS.

- The aforementioned construction of the `CertificateVerify` requires on average about 28000 cycles. These cycles are almost exclusively spend on computing the SS-Base-Key and the MAC as was shown in Figure 3.4.

On the client side there is a difference of about 50000 in OPTLS' favor. The difference can be explained as follows:

- In `ssl_derive` for the OPTLS we can not generate the static secret yet since we do not have the server's public key yet. However in the Internet Draft the client can generate up to the handshake secret since g^{xs} is not yet required. This means that `ssl_derive` is about 22000 cycles faster for OPTLS.
- Lastly for OPTLS we derive g^{xs} and generate the static secret; this happens in `tls_process_server_certificate` and requires about 176000 cycles, of which 168000 are spent deriving g^{xs}. For the Internet Draft more cycles are required: the total for `tls_process_cert_verify` comes to about 204000, again 16800 of those are spent deriving g^{xs} and like the client it takes about 28000 cycles to compute the SS-Base-Key and MAC.

Constructing the `CertificateVerify` takes 28000 cycles. We thought this was high for only symmetric operations, most of which should be hashing. We looked further into where exactly these cycles are being spent, we identified four major operations:

1. Deriving the SS-Base-Key takes 7000 cycles.
2. Then deriving the finishedkey also takes 7000 cycles.
3. Creating the actual MAC key from the finishedkey, i.e calling `EVP_PKEY_new_raw_private_key`, takes about 2200 cycles.
4. Finally computing the MAC takes 11000 cycles.

We expected most of these cycles to be spent computing SHA256. To verify this we removed the inner compression function of SHA256 and reran our benchmarks. With this part of the algorithm removed the whole construction still takes around 21000 cycles; 5800 for the SS-Base-Key, 5000 for the finishedkey, still 2200 for the actual MAC

key and 7800 to compute the MAC. Apparently there is quite some overhead.

A final interesting observation is the poor performance of X448, and the NIST curves P-384 and P-521. We speculate that the implementation of these algorithms could be further optimized, interestingly enough P-521 does have some optimizations which P-384 does not, making it faster. Thus — although every certificate type offers the advantages of OPTLS described in Chapter 3 — we recommend X25519 for optimal performance of OPTLS.

Chapter 6

A look into the future

So far we have delivered an implementation of OPTLS and explored its performance. However, the main cryptographic primitive we have used is ECDH. This means neither variant of OPTLS offers protection against quantum computers. Initially we set out to integrate arbitrary KEMs into TLS to provide this post-quantum security, however further research is needed to turn this into reality. In this chapter we discuss the current state of post-quantum TLS and state our contribution.

6.1 Post-quantum TLS

In the future the world will need to switch its cryptographic primitives to be resistant to attacks to a quantum computer, but what does this mean? We have seen that a protocol such as TLS uses several primitives to establish the secure channel over which the application data is sent. The things that need to be replaced are the cryptographic algorithms in the bottom layer. We have for instance used the Diffie-Hellman key exchange to establish a shared secret. For authentication we used digital signatures, which are assumed to be distributed by some central authority establishing identities. To derive cryptographic keys we have used a hash-based scheme using HMACs. Lastly to protect the actual contents of the messages we use symmetric cryptography. In many cases we need to reevaluate the choice of algorithm.

6.1.1 Security of primitives

An attacker with access to a large quantum computer can break the security offered by some of the schemes mentioned in Section 6.1. For one solving the discrete logarithm can be done in polynomial time on such a machine; the solution can be found using Shor's algorithm [41]. With DH being so fundamental in OPTLS, this seems like a major issue in making OPTLS quantum resistant. For digital signatures there are several post-quantum replacements available [42, 43, 44, 45, 46]; hybrid solutions have also been considered [47]. However post-quantum signatures have different (i.e usually worse) performance characteristics [48]. Moreover, key exchanges today can be recorded and there can be a real risk if these are broken in twenty years. In contrast, signatures are usually immediately verified to authenticate, even if the private signing key would be recovered much later using a quantum computer it may have expired anyway.

In the case of hashing and symmetric cryptography the situation appears to be not as bad. For symmetric ciphers an exhaustive search of the key space can be accomplished in the square root of the size of the space using Grover's algorithm [49]. This means that doubling the key size — moving from AES-128 to AES-256 — should be enough to retain the same security level. But note that recently some doubt has been cast on the security of certain block cipher constructions against quantum attackers [50, 51].

6.1.2 Efforts so far

Since the trust in security of algorithms is only established after resisting attacks for a long time it makes sense to move towards post-quantum TLS using a *hybrid* key exchange using both classic and post-quantum algorithms. This remains secure as long as one of the selected algorithms is not broken. This has several advantages: offering post-quantum security early, maintaining compatibility with standards and reducing risk from the uncertainty of post-quantum algorithms.

Such an experiment was performed by Google for TLS 1.2 using the NewHope [52] KEM. For this they combined NewHope with X25519. No unexpected issues were found enabling this key exchange, although the larger message sizes did increase the latency [53].

Another significant contribution to post-quantum TLS is OpenQuan-

tumSafe [54] library. This library integrates into OpenSSL to provide post-quantum primitives for use with TLS 1.2. This allows one to test and benchmark quantum resistant algorithms. However this is for TLS 1.2 and our focus was on modifying TLS 1.3 to make it more amendable towards using a KEM, thus we did not use this library.

6.1.3 Post-quantum Signatures

We do not exhaustively treat post-quantum signature schemes, this section is intended as a quick overview.

Note that in TLS 1.3 there are two different "types" of signatures to be considered, the first is the signature provided by the CA. This signature is on the long-term public key of a server (the *certificate*). This is the only signature that is still required in OPTLS. Since the signature needs to be sent by the server for every handshake a proper candidate should first most have small signature sizes and secondly fast verification times since the client, which may have limited resources, needs to verify every time. Less important is the signing time since this is done only once by a CA. We may even cautiously consider stateful signature schemes; perhaps a CA could be assumed to keep state. Even today there already are standardized algorithms which could accomplish this, see XMSS [42].

More difficult — hence our interest in OPTLS — is an online signature that a server provides on every handshake. This additionally requires fast signing time and stateful schemes become too unreliable to be practical.

We roughly divide the available (non-stateful) schemes into categories, numbers are from [48], we do not care (within reason) much for key generation time:

Hash-based Such as Gravity-SPHINCS and SPHINCS$^+$, these come with public keys of no more than 64 bytes. However signature size is somewhat large, starting at 8 kB and becoming larger as a trade-of for signing speed and security level. Verification is also somewhat slow with 7 million cycles minimum for SPINCS$^+$. Note that these schemes come with a maximum amount of signatures for one private key, although this amount is high enough that in general it will not be reached.

Lattice-based For example CRYSTALS-DILITHIUM and qTesla. Public keys here start at 1 kB, and signatures at 2 kB. However

signing and verification is very fast. This appears to be the most balanced approach.

Multivariate Contenders are MQDSS, LUOV and others. MQDSS has small public and secret keys, respectively 64 and 24 bytes. With 34 kB the size of signatures is rather large though. LUOV has significantly larger public keys (more than 15 kB) but enjoys much smaller signatures (around 3 kB). However these are also slow.

Code-based We are aware of RaCoSS, pqsigRM and RankSign. The last one was already withdrawn from the NIST competition and the other two appear to be under heavy attack.

Isogeny-based Signature sizes are over 100 kB, making this unpractical, no submissions to NIST are based on this.

6.2 Using KEMs

In the literature on OPTLS we found some indications that a standard Diffie-Hellman key exchange can be replaced with a KEM [55, 56]. However we stress that a Diffie-Hellman key exchange can in general not be replaced by a KEM based AKE. Even for our objective of obtaining a UAKE there are certain conditions. In this section we will look into what we can build using a KEM as our main cryptographic primitive, while limiting ourselves to a maximum of one round trip; keeping in mind that this was one of the requirements of TLS 1.3.

6.2.1 Variant 1

The first variant we consider is illustrated in Figure 6.1. In this scheme the client initiates the connection and sends it's public key PK_C, signed by a CA over to the server. The server now knows that C *might* be trying to authenticate, thus it uses PK_C to generate a ciphertext ct_{K_1}, which *encapsulates* a secret K_1. Note that the server is not sure yet if it is really talking to C, but we assume it wants to authenticate regardless and thus includes PK_S, also signed by a CA. The client receives PK_S, and ct_{K_1}; it can recover K_1 using its private key, but still has no guarantee that it is actually talking to S. Thus is generates a ciphertext ct_{K_2} and sends that together with a MAC over the handshake transcript, for which both keys are used.

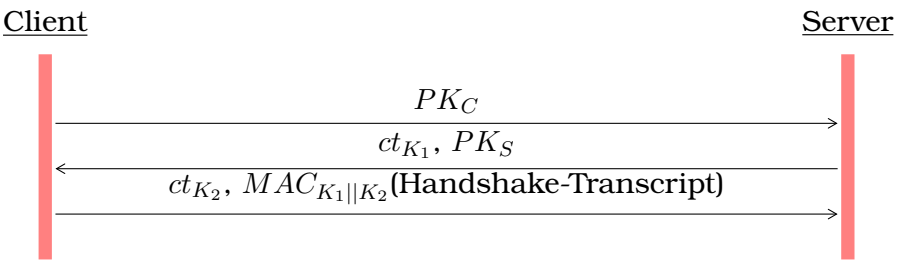

Figure 6.1: KEM authentication of client

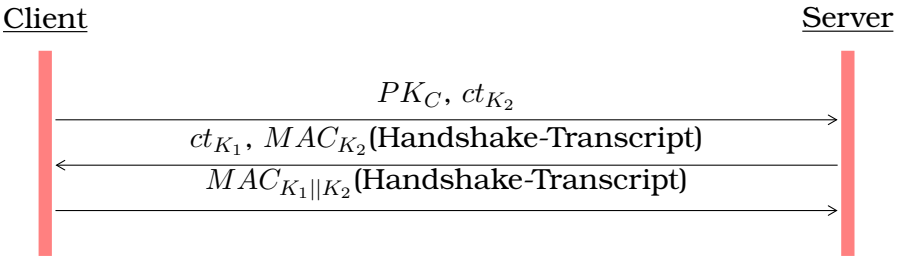

Figure 6.2: Possible mutual KEM authentication, assuming pre-distributed server public keys

Now the server can verify everything went correctly using its private key. However the server was **not** authenticated to the client yet. Thus the client is not allowed to include application data in the second round trip; failing the requirements of TLS 1.3. The client is authenticated to the server though, which might be useful, although there is another incompatibility making this unfit for TLS 1.3; the certificates are not encrypted thus the endpoints' identities are not protected [57].

6.2.2 Variant 2

To fix the scheme from Section 6.2.1 we assume the client already knows the server's public key. We then obtain the authentication protocol depicted in Figure 6.2.

The protocol looks similar, the major difference being in the first message sent, it now includes a ciphertext encapsulating the key K_2, which is obtained using the server's public key. Now only the legitimate server is able to prove possession of the private key, thus authenticating it to the client. The server can also optionally issue a challenge to the client to authenticate. When the client now validates

the MAC, it knows the server must possess the private key, and by the CA's signature thus knows its identity. Therefore application data can be included in the next round trip.

Unfortunately neither of the two variants accomplish our objective of a 1-RTT (server) UAKE in the setting of TLS 1.3. The first variant ends the key exchange with an authenticated client, but without an authenticated server. This is a major issue given that we are more interested in server authentication. The second variant requires pre-distributed public keys. It is not clear how this would be achieved, in fact if it could be achieved then data could even be sent in the first flight, although with no forward secrecy or guarantee of liveliness.

The issue we have identified is thus as follows:

- To achieve an OPTLS based (server) UAKE the server needs to actually use its private key in a cryptographic operation for the first response. If only the public key is used then the server does not authenticate to the client. In OPTLS this is achieved by creating the Diffie-Hellman shared secret g^{xs} on both sides. The client knows when it receives g^s that to obtain the same shared secret the server *must* have s. In contrast for the KEM based authentication proving possession of the secret key *requires interaction*; the server *must* decapsulate the provided ciphertext to authenticate.

Thus we stress that an interactive key exchange primitive can not be used in OPTLS as specified in the original proposal. The Diffie-Hellman key exchange can in this protocol only be replaced by another NIKE.

6.3 The way forward

We have shown that it is not possible to use an interactive key exchange primitive to authenticate a server in a 1-RTT protocol initiated by the client. In this section we list some possible alternatives, unfortunately given the limitations only the first one makes use of KEMs.

TLS 1.2 style

The first option, which retains KEMs is to instead use a TLS 1.2 style handshake protocol. This would look very much like Figure 6.1, except that the client would send its certificate optionally only in the second message. This is what liboqs already provides, but has the disadvantage of more latency because of the 2-RTT required before the handshake is complete, we feel for such a major internet protocol a 1-RTT key exchange is worth striving for.

TLS 1.3 style

To do a TLS 1.3 style handshake with post-quantum algorithms we need a replacement for the ephemeral Diffie-Hellman key exchange and a signature algorithm. Ephemeral (but **not** static [58]) Diffie-Hellman keys could be replaced by Supersingular Isogeny Diffie-Hellman (SIDH) [59], which allows for public keys of 330 bytes [60], at least comparable to ECDH. The amount of clock cycles required for this key exchange would unfortunately be orders of magnitude larger than for current ECDH. This can then be used together with a post-quantum signature scheme to obtain an authentication scheme similar to SIGMA [61, 62].

OPTLS style

The final approach we discuss would be to build an algorithm with security properties equivalent to Diffie-Hellman in a post-quantum setting. As mentioned this requires a post-quantum NIKE [29]. Currently there are no standardized candidates, although research into these algorithms is ongoing. We are aware of one SIDH based algorithm [63], which is unpractical, and one recent experimental candidate: CSIDH [64], based on the class group action on supersingular curves.

If CSIDH or a similar (NIKE) algorithm is "found secure" and standardized we expect no problems using it for a variant of OPTLS. The Internet Draft would be a good candidate as new algorithms can easily be added through use of the `sig_algs` extension.

Chapter 7

Conclusion

In this thesis we have described some of the inner working of OpenSSL to hopefully help with future implementations and we built on this ourselves to provide an implementation of two different styles of OPTLS. We then benchmarked these and discussed some interesting aspects regarding the performance of our implementations and we provided numbers which allow a practical comparison between the two styles and TLS 1.3. We also contribute a reminder that a KEM can not replace a NIKE in general while discussing possible directions for post-quantum TLS.

7.1 A note on security proofs

While OPTLS comes with a security proof and TLS 1.3 has been extensively scrutinized by the cryptographic community [65], the Internet Draft has not (yet) received similar attention. This is not unexpected as it is in an early stage. However caution is advised using this protocol in the state it is in now. It would be interesting to see if the security proof of OPTLS can be adapted for this Internet Draft.

7.2 Future work

Apart from the aforementioned security proof some interesting problems remain.

- Iterating on the Internet Draft we implemented will be interesting, we provided an implementation of the `-00` version, but we did not look to improve it. Speed optimizations may be possible by looking more in depth at what security properties the MAC key for the `CertificateVerify` key exactly needs to provide.
- Integrating KEMs in some other way, since the NIST not-a-competition will eventually provide us with a standardized quantum resistant KEM it would be very interesting if we can somehow further optimize the 2-RTT TLS 1.2 style protocol.
- Optimizing post-quantum schemes, this may be obvious, but any speed gains or size reductions in the available post-quantum algorithms will directly translate to performance improvements. If these algorithms are used in the TLS handshake protocol it will also directly influence which handshake variant is optimal.
- Research into a post-quantum NIKE, we already stated the only option at the moment is the experimental CSIDH. Either establishing confidence in this algorithm or coming up with a new NIKE is a requirement for post-quantum OPTLS.

7.3 Final words

We have shown for two variants of OPTLS that they can be practically implemented into OpenSSL. We have delivered two implementations of variants of OPTLS integrated in OpenSSL.

However we have also seen that more research is required before a post-quantum version of TLS can be standardized. While a post-quantum NIKE would solve our problems with building post-quantum OPTLS it is not immediately obvious if such an algorithm will be available in the future. Looking into non-obvious ways to integrate KEMs may be more successful short term.

Bibliography

[1] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Internet-Draft draft-ietf-tls-tls13-28, Internet Engineering Task Force, 2018. https://datatracker.ietf.org/doc/html/draft-ietf-tls-tls13-28. 4

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978. https://people.csail.mit.edu/rivest/Rsapaper.pdf. 4

[3] H. Krawczyk and H. Wee, "The OPTLS Protocol and TLS 1.3," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 81–96, IEEE, 2016. https://eprint.iacr.org/2015/978.pdf. 5, 24

[4] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976. http://www-ee.stanford.edu/~hellman/publications/24.pdf. 5

[5] M. D. Raimondo, R. Gennaro, and H. Krawczyk, "Deniable Authentication and Key Exchange," in *13th ACM Conference on Computer and Communications Security — CCS'06*, pp. 400–409, ACM, 2006. https://eprint.iacr.org/2006/280. 5

[6] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001. https://doi.org/10.1007/s102070100002. 5

[7] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012. https://link.springer.com/content/pdf/10.1007/s13389-012-0027-1.pdf. 5

[8] E. Rescorla and N. Sullivan, "Semi-Static Diffie-Hellman Key Establishment for TLS 1.3," Internet-Draft draft-rescorla-tls13-semistatic-dh-00, Internet Engineering Task Force, 2018. https://datatracker.ietf.org/doc/html/draft-rescorla-tls13-semistatic-dh-00. 6, 8, 24

[9] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987. http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/. 6

[10] V. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology — CRYPTO'85*, vol. 218 of *Lecture Notes in Computer Science*, pp. 417–426, Springer, 1986. https://link.springer.com/content/pdf/10.1007/3-540-39799-X_31.pdf. 6

[11] IBM, "IBM scientists achieve critical steps to building first practical quantum computer." Press release, 2015. https://ibm.com/press/us/en/pressrelease/46725.wss, retrieved 24-09-18. 6

[12] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on Post-Quantum Cryptography," NISTIR 8105, National Institute for Standards and Technology, 2016. https://doi.org/10.6028/NIST.IR.8105. 6

[13] S. Rich and B. Gellman, "NSA seeks to build quantum computer that could crack most types of encryption," 2014. The Washington Post. 6

[14] NIST, "Post-quantum crypto project," 2016. https://csrc.nist.gov/projects/post-quantum-cryptography, retrieved 24-09-18. 6

[15] R. Cramer and V. Shoup, "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack," *SIAM Journal on Computing*, vol. 33, no. 1, pp. 167–226, 2003. http://shoup.net/papers/cca2.pdf. 6, 12

[16] W. Diffie, P. C. V. Oorschot, and M. J. Wiener, "Authentication and authenticated key exchanges," *Designs, Codes and Cryptography*, vol. 2, no. 2, pp. 107–125, 1992. http://people.scs.carleton.ca/~paulv/papers/sts-final.pdf. 10

[17] M. Bellare and P. Rogaway, "Entity Authentication and Key Distribution," in *Advances in Cryptology — CRYPTO'93*, vol. 733

of *Lecture Notes in Computer Science*, pp. 232–249, Springer, 1994. https://cseweb.ucsd.edu/~mihir/papers/eakd.pdf. 10

[18] Y. Dodis and D. Fiore, "Unilaterally-Authenticated Key Exchange," in *Financial Cryptography and Data Security*, vol. 10322 of *Lecture Notes in Computer Science*, pp. 542–560, Springer, 2017. https://eprint.iacr.org/2017/109.pdf. 11

[19] U. M. Maurer and S. Wolf, "The Diffie–Hellman Protocol," *Designs, Codes and Cryptography, Special Issue Public Key Cryptography*, vol. 19, no. 3, pp. 147–171, 2000. ftp://ftp.inf.ethz.ch/pub/crypto/publications/MauWol00c.pdf. 11

[20] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," in *22nd ACM Conference on Computer and Communications Security — CCS'15*, pp. 5–17, ACM, 2015. https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf. 11

[21] M. Musson, "Attacking the Elliptic Curve Discrete Logarithm Problem," Master's thesis, Acadia University, 2006. https://www.researchgate.net/file.PostFileLoader.html?id=5583faed6225ff040e8b458b&assetKey=AS%3A273804985602059%401442291603545. 11

[22] D. J. Bernstein and T. Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography." https://safecurves.cr.yp.to, retrieved 10-09-18. 12

[23] NIST, "Digital Signature Standard," FIPS PUBS 186-2, National Institute for Standards and Technology, 2000. http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf. 12

[24] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," in *Public Key Cryptography — PKC'06*, vol. 3958 of *Lecture Notes in Computer Science*, pp. 207–228, Springer, 2006. https://cr.yp.to/ecdh/curve25519-20060209.pdf. 12

[25] M. Hamburg, "Ed448-Goldilocks, a new elliptic curve." Cryptology ePrint Archive, Report 2015/625, 2015. https://eprint.iacr.org/2015/625. 12

[26] C. G. Günther, "An Identity-Based Key-Exchange Protocol," in *Advances in Cryptology — EUROCRYPT'89*, vol. 434 of *Lecture*

Notes in Computer Science, pp. 29–37, Springer, 1990. https://link.springer.com/content/pdf/10.1007/3-540-46885-4_5.pdf. 12

[27] C. Adams, G. Kramer, S. Mister, and R. Zuccherato, "On The Security of Key Derivation Functions," in *Information Security — ISC'04*, vol. 3225 of *Lecture Notes in Computer Science*, pp. 134–145, Springer, 2004. https://link.springer.com/chapter/10.1007/978-3-540-30144-8_12. 12

[28] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," in *Advances in Cryptology — CRYPTO'84*, vol. 196 of *Lecture Notes in Computer Science*, pp. 10–18, Springer, 1985. https://link.springer.com/content/pdf/10.1007/3-540-39568-7_2.pdf. 13

[29] E. S. V. Freire, D. Hofheinz, E. Kiltz, and K. G. Paterson, "Non-Interactive Key Exchange," in *Public-Key Cryptography — PKC'13*, vol. 7778 of *Lecture Notes in Computer Science*, pp. 254–271, Springer, 2013. https://eprint.iacr.org/2012/732.pdf. 13, 14, 52

[30] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Advances in Cryptology — CRYPTO'98*, vol. 1462 of *Lecture Notes in Computer Science*, pp. 26–45, Springer, 1998. https://link.springer.com/content/pdf/10.1007/BFb0055718.pdf. 14

[31] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology — CRYPTO'96*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 1–15, Springer, 1996. https://link.springer.com/content/pdf/10.1007/3-540-68697-5_1.pdf. 16

[32] H. Krawczyk, "Cryptographic Extraction and Key Derivation: The HKDF Scheme," in *Advances in Cryptology —- CRYPTO'10*, vol. 7778 of *Lecture Notes in Computer Science*, pp. 254–271, Springer, 2010. https://link.springer.com/content/pdf/10.1007/978-3-642-14623-7_34.pdf. 16

[33] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, RFC Editor, 1999. http://www.rfc-editor.org/rfc/rfc2246.txt. 18

[34] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, RFC Editor, 2006. http://www.rfc-editor.org/rfc/rfc4346.txt. 18

[35] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, RFC Editor, 2008. http://www.rfc-editor.org/rfc/rfc5246.txt. 18

[36] E. Rescorla, "Update on TLS 1.3 Middlebox Issues." on the IETF mailing list, 2017. https://www.ietf.org/mail-archive/web/tls/current/msg24517.html. 20

[37] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Internet-Draft draft-ietf-tls-tls13-09, Internet Engineering Task Force, 2015. http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-09.txt. 24

[38] F. Günther, *Modeling Advanced Security Aspects of Key Exchange and Secure Channel Protocols*. PhD thesis, Technische Universität Darmstadt, 2018. http://tuprints.ulb.tu-darmstadt.de/7162/. 26

[39] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," RFC 5869, RFC Editor, 2010. http://www.rfc-editor.org/rfc/rfc5869.txt. 26

[40] H. Krawczyk, "OPTLS: Signature-less TLS 1.3." on the IETF mailing list, 2015. https://www.ietf.org/mail-archive/web/tls/current/msg14385.html. 27

[41] P. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Review*, vol. 41, no. 2, pp. 303–332, 1999. https://arxiv.org/pdf/quant-ph/9508027.pdf. 47

[42] A. Huelsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme," RFC 8391, RFC Editor, 2018. https://rfc-editor.org/rfc/rfc8391.txt. 47, 48

[43] M.-S. Chen, A. Hülsing, J. Rijneveld, S. Samardjiska, and P. Schwabe, "From 5-pass MQ-based identification to MQ-based signatures," in *Advances in Cryptology — ASIACRYPT'16*, vol. 10032 of *Lecture Notes in Computer Science*, pp. 135–165, Springer, 2016. https://eprint.iacr.org/2016/708. 47

[44] L. Ducas, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS – Dilithium: Digital signatures from module lattices," *Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 1, pp. 238–268, 2018. https://eprint.iacr.org/2017/633.pdf. 47

[45] D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, and P. Schwabe, "SPHINCS$^+$: Submission to the NIST post-quantum project." Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. https://cryptojedi.org/papers/#sphincsnist. 47

[46] J.-P. Aumasson and G. Endignoux, "Improving Stateless Hash-Based Signatures," in *Topics in Cryptology — CT-RSA 2018*, vol. 10808 of *Lecture Notes in Computer Science*, pp. 219–242, Springer, 2018. https://eprint.iacr.org/2017/933.pdf. 47

[47] N. Bindel, U. Herath, M. McKague, and D. Stebila, "Transitioning to a Quantum-Resistant Public Key Infrastructure," in *Post-Quantum Cryptography — PQCRYPTO'17*, vol. 10346 of *Lecture Notes in Computer Science*, pp. 384–405, Springer, 2017. https://eprint.iacr.org/2017/460.pdf. 47

[48] SAFECrypto. https://www.safecrypto.eu/pqclounge/software-analysis-signatures/, retrieved 11-09-18. 47, 48

[49] L. K. Grover, "A Fast Quantum Mechanical Algorithm for Database Search," in *Annual ACM Symposium on Theory of Computing — STOC'96*, pp. 212–219, ACM, 1996. https://arxiv.org/pdf/quant-ph/9605043. 47

[50] X. Dong, Z. Li, and X. Wang, "Quantum cryptanalysis on some Generalized Feistel Schemes," *SCIENCE CHINA Information Sciences*, 2017. https://eprint.iacr.org/2017/1249. 47

[51] X. Bonnetain and M. Naya-Plasencia, "Hidden Shift Quantum Cryptanalysis and Implications." Cryptology ePrint Archive, Report 2018/432, 2018. https://eprint.iacr.org/2018/432. 47

[52] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange – a new hope," in *25th USENIX Security Symposium — USENIX Security'16*, pp. 327–343, USENIX Association, 2016. https://cryptojedi.org/papers/newhope-20171212.pdf. 47

[53] A. Langley, "Post-quantum confidentiality for TLS," 2016. https://www.imperialviolet.org/2016/11/28/cecpq1.html. 47

[54] D. Stebila and M. Mosca, "Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project," in *Selected Areas in Cryptography — SAC'16*, vol. 10532 of *Lecture Notes in*

Computer Science, pp. 14–37, Springer, 2017. https://eprint.iacr.org/2016/1017. 48

[55] H. Krawczyk, "An Introducion to the Design and Analysis of Key Exchange Protocols." Presentation, 2018. https://cyber.biu.ac.il/wp-content/uploads/2018/07/KE1_Hugo_BIU_Feb2018-online.pdf. 49

[56] P. Schwabe, "The transition to post-quantum cryptography." Presentation, 2018. https://cryptojedi.org/peter/data/nancy-20180219.pdf. 49

[57] M. Wachs, Q. Scheitle, and G. Carle, "Push away your privacy: Precise user tracking based on TLS client certificate authentication," in *Network Traffic Measurement and Analysis — TMA'17*, IEEE, 2017. http://tma.ifip.org/wordpress/wp-content/uploads/2017/06/tma2017_paper2.pdf. 50

[58] S. D. Galbraith, C. Petit, B. Shani, and Y. B. Ti, "On the Security of Supersingular Isogeny Cryptosystems," in *Advances in Cryptology — ASIACRYPT'16*, vol. 10031 of *Lecture Notes in Computer Science*, pp. 63–91, Springer, 2016. https://eprint.iacr.org/2016/859.pdf. 52

[59] L. D. Feo, D. Jao, and J. Plût, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," *Journal of Mathematical Cryptology*, vol. 8, no. 3, p. 209–247, 2014. http://eprint.iacr.org/2011/506.pdf. 52

[60] C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes, and D. Urbanik, "Efficient Compression of SIDH Public Keys," in *Advances in Cryptology — EUROCRYPT'17*, vol. 10210 of *Lecture Notes in Computer Science*, pp. 679–706, Springer, 2017. https://eprint.iacr.org/2016/963.pdf. 52

[61] P. Longa, "A Note on Post-Quantum Authenticated Key Exchange from Supersingular Isogenies." Cryptology ePrint Archive, Report 2018/267, 2018. https://eprint.iacr.org/2018/267.pdf. 52

[62] H. Krawczyk, "SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols," in *Advances in Cryptology — CRYPTO'03*, vol. 2729 of *Lecture Notes in Computer Science*, pp. 400–425, Springer, 2003. https://link.springer.com/content/pdf/10.1007/978-3-540-45146-4_24.pdf. 52

[63] R. Azarderakhsh, D. Jao, and C. Leonardi, "Post-Quantum Static-Static Key Agreement Using Multiple Protocol Instances," in *Selected Areas in Cryptography — SAC'17*, vol. 10719 of *Lecture Notes in Computer Science*, pp. 45–63, Springer, 2018. http://www.site.uottawa.ca/~cadams/papers/prepro/paper_31.pdf. 52

[64] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes, "CSIDH: An Efficient Post-Quantum Commutative Group Action," in *Advances in Cryptology — ASIACRYPT'18*, Springer, forthcoming. https://eprint.iacr.org/2018/383. 52

[65] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 handshake protocol candidates," in *22nd ACM Conference on Computer and Communications Security — CCS'15*, pp. 1197–1210, ACM, 2015. https://eprint.iacr.org/2015/914.pdf. 53

Appendix A

Protocol traces OPTLS

```
Using default temp DH parameters
ACCEPT [::]:4433
Received Record
Header:
  Version = TLS 1.0 (0x301)
  Content Type = Handshake (22)
  Length = 224
    ClientHello, Length=220
      client_version=0x303 (TLS 1.2)
      Random:
        gmt_unix_time=0xC3771CC3
        random_bytes (len=28):
    AEAFEFD17D00BB062DC45BE56C68D5FC5FBD96D1BADEC20AB094C862
      session_id (len=32): 452
    E4BA623C57B09A2F1C2DCAA82C33FEA0BD188C63FC48881E2311B30F5DB20

      cipher_suites (len=4)
        {0x13, 0x03} TLS_CHACHA20_POLY1305_SHA256
        {0x00, 0xFF} TLS_EMPTY_RENEGOTIATION_INFO_SCSV
      compression_methods (len=1)
        No Compression (0x00)
      extensions, length = 143
        extension_type=server_name(0), length=14
          0000 - 00 0c 00 00 09 6c 6f 63-61 6c 68 6f 73 74
     .....localhost
        extension_type=ec_point_formats(11), length=4
          uncompressed (0)
          ansiX962_compressed_prime (1)
          ansiX962_compressed_char2 (2)
        extension_type=supported_groups(10), length=12
          ecdh_x25519 (29)
          secp256r1 (P-256) (23)
          ecdh_x448 (30)
          secp521r1 (P-521) (25)
          secp384r1 (P-384) (24)
```

```
        extension_type=session_ticket(35), length=0
        extension_type=encrypt_then_mac(22), length=0
        extension_type=extended_master_secret(23), length=0
        extension_type=signature_algorithms(13), length=30
          ecdsa_secp256r1_sha256 (0x0403)
          ecdsa_secp384r1_sha384 (0x0503)
          ecdsa_secp521r1_sha512 (0x0603)
          ed25519 (0x0807)
          ed448 (0x0808)
          rsa_pss_pss_sha256 (0x0809)
          rsa_pss_pss_sha384 (0x080a)
          rsa_pss_pss_sha512 (0x080b)
          rsa_pss_rsae_sha256 (0x0804)
          rsa_pss_rsae_sha384 (0x0805)
          rsa_pss_rsae_sha512 (0x0806)
          rsa_pkcs1_sha256 (0x0401)
          rsa_pkcs1_sha384 (0x0501)
          rsa_pkcs1_sha512 (0x0601)
        extension_type=supported_versions(43), length=3
          OPTLS (773)
        extension_type=psk_key_exchange_modes(45), length=2
          psk_dhe_ke (1)
        extension_type=key_share(51), length=38
            NamedGroup: ecdh_x25519 (29)
            key_exchange:  (len=32): 9
    BBD78EE56D101EEAF863EB928D4EA6851FB93A4E5FFD58FE28A6E82D41D267B


Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = Handshake (22)
  Length = 122
    ServerHello, Length=118
      server_version=0x303 (TLS 1.2)
      Random:
        gmt_unix_time=0xA4E611BE
        random_bytes (len=28): 48
    A6190CAC8BAB58645BB56145CED89E032F3EFE0444A0CB4B712A8B
      session_id (len=32): 452
    E4BA623C57B09A2F1C2DCAA82C33FEA0BD188C63FC48881E2311B30F5DB20

      cipher_suite {0x13, 0x03} TLS_CHACHA20_POLY1305_SHA256
      compression_method: No Compression (0x00)
      extensions, length = 46
        extension_type=supported_versions(43), length=2
            OPTLS (773)
        extension_type=key_share(51), length=36
            NamedGroup: ecdh_x25519 (29)
            key_exchange:  (len=32): 463
    C26126CA9F99D9BC43DC3E502DB7387FA3046C6A9EBA5A015DDC1A33C2632


Sent Record
```

```
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ChangeCipherSpec (20)
  Length = 1
    change_cipher_spec (1)

Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 23
  Inner Content Type = Handshake (22)
    EncryptedExtensions, Length=2
      extensions, length = 0

Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 359
  Inner Content Type = Handshake (22)
    Certificate, Length=338
      context (len=0):
      certificate_list, length=334
        ASN.1Cert, length=329
------details-----
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            f5:d1:34:cd:4c:4f:76:7f
        Signature Algorithm: ED25519
        Issuer: C = NL, ST = Some-State, L = Nijmegen, O = RU,
   OU = ICIS
        Validity
            Not Before: Feb 24 22:09:10 2018 GMT
            Not After : Mar 26 22:09:10 2018 GMT
        Subject: C = AU, ST = Some-State, O = Internet Widgits
   Pty Ltd
        Subject Public Key Info:
            Public Key Algorithm: X25519
                X25519 Public-Key:
                pub:
                  7e:44:aa:5f:ce:67:c7:71:27:a7:b6:66:98:55:73:
                  8c:72:ea:4d:f4:a7:23:0b:ab:9e:92:80:32:bb:d9:
                  91:65
    Signature Algorithm: ED25519
         08:5a:9b:bd:a3:04:b3:bd:a4:d7:06:49:6a:5b:33:93:13:8f:
         02:4f:8c:c2:0d:6b:22:72:35:9e:b9:00:29:7b:99:7f:a5:d6:
         4b:e4:d9:94:8e:ee:bd:aa:ac:4a:23:15:e2:bf:19:e5:aa:c2:
         6d:ca:8e:ae:f9:79:cd:39:80:07
-----BEGIN CERTIFICATE-----
MIIBRTCB+AIJAPXRNM1MT3Z/MAUGAytlcDBRMQswCQYDVQQGEwJ
OTDETMBEGA1UECAwKU29tZS1TdGF0ZTERMA8GA1UEBwwITmlqbW
```

```
VnZW4xCzAJBgNVBAoMAlJVMQ0wCwYDVQQLDARJQ0lTMB4XDTE4M
DIyNDIyMDkxMFoXDTE4MDMyNjIyMDkxMFowRTELMAkGA1UEBhMC
QVUxEzARBgNVBAgMClNvbWUtU3RhdGUxITAfBgNVBAoMGEludGV
ybmV0IFdpZGdpdHMgUHR5IEx0ZDAqMAUGAytlbgMhAH5Eql/OZ8
dxJ6e2ZphVc4xy6k30pyMLq56SgDK72ZFlMAUGAytlcANBAAham
72jBLO9pNcGSWpbM5MTjwJPjMINayJyNZ65ACl7mX+l1kvk2ZSO
7r2qrEojFeK/GeWqwm3Kjq75ec05gAc=
-----END CERTIFICATE-----
------------------
        extensions, length = 0

Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 53
  Inner Content Type = Handshake (22)
    Finished, Length=32
      verify_data (len=32): 17
    AAD4A398BC8A8FF637005DF3B58422687E4E81626781978602814EED5C2B25


Received Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ChangeCipherSpec (20)
  Length = 1
Received Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 53
  Inner Content Type = Handshake (22)
    Finished, Length=32
      verify_data (len=32): 2
    E841F5ACCAAB93F7F2A0CEEB728D47BEB593801349C6607EEA1961E7BBE1286


Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 243
  Inner Content Type = Handshake (22)
    NewSessionTicket, Length=222
        ticket_lifetime_hint=7200
        ticket_age_add=813870700
        ticket_nonce (len=1): 00
        ticket (len=208):
        D797460547BB80802E2C828769A35A11635C3E9198279170499
        22203F068C9B9C629442F2DF68868B2F566B8BE46889EB5ECB9
        59AA7D4BB3D9CC7A222690F3117608DDDE4E46FDDCB6BF21AB8
        210D1412FBCF5F1F8E7B506E1675055C993966CF1D7D64AA676
        240B84C77019F0CC54D683EC20A7D08F4F52EC5F1F7CACF773F
```

```
        B7FDD63EEAE0D4D5D35A8C7809DCF116045A0DCA12FC3A352FF
        706F47545D6A3A9A432ECBB96F8EDB3EF883119624E043F95D4
        DE6018FCCBC020A1C8EE36EC20243C992DD6DAD90CD30F14591
        CBEEE3DD
        extensions, length = 0

-----BEGIN SSL SESSION PARAMETERS-----
MH8CAQECAgMFBAITAwQgO6PFpOWrok9uwcg2C5SdRiWm0ZjMTu9
NCS+27ehD/VQEIDhzaV4/HufhKFyz6JvRJoX1RHRSeKSsMaXV9d
+eaASQoQYCBFt2xGiiBAICHCCkBgQEAQAAAKYLBAlsb2NhbGhvc
3SuBgIEMIKubLEDBAEA
-----END SSL SESSION PARAMETERS-----
Shared ciphers:TLS_CHACHA20_POLY1305_SHA256
Signature Algorithms: ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA512:
    Ed25519:Ed448:RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA512:
    RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA512:RSA+SHA256:RSA
    +SHA384:RSA+SHA512
Shared Signature Algorithms: ECDSA+SHA256:ECDSA+SHA384:ECDSA+
    SHA512:Ed25519:Ed448:RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+
    SHA512:RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA512:RSA+
    SHA256:RSA+SHA384:RSA+SHA512
Supported Elliptic Groups: X25519:P-256:X448:P-521:P-384
Shared Elliptic groups: X25519:P-256:X448:P-521:P-384
---
No server certificate CA names sent
CIPHER is TLS_CHACHA20_POLY1305_SHA256
Secure Renegotiation IS supported
Received Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 19
  Inner Content Type = Alert (21)
    Level=warning(1), description=close notify(0)

DONE
```

Appendix B

Protocol traces Internet Draft

```
Using default temp DH parameters
ACCEPT
Received Record
Header:
  Version = TLS 1.0 (0x301)
  Content Type = Handshake (22)
  Length = 308
    ClientHello, Length=304
      client_version=0x303 (TLS 1.2)
      Random:
        gmt_unix_time=0x2465489B
        random_bytes (len=28): 84
    E1A80B7192552C16EFBC5C5BC1DBD948CC5082D12E71827C74C758
      session_id (len=32):
    FAFE6D982F9FEABAE5767008249547DCECE39BBD58379A58CA6335E8ADA25259

      cipher_suites (len=62)
        {0x13, 0x02} TLS_AES_256_GCM_SHA384
        {0x13, 0x03} TLS_CHACHA20_POLY1305_SHA256
        {0x13, 0x01} TLS_AES_128_GCM_SHA256
        {0xC0, 0x2C} TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
        {0xC0, 0x30} TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
        {0x00, 0x9F} TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
        {0xCC, 0xA9}
    TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
        {0xCC, 0xA8}
    TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
        {0xCC, 0xAA} TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
        {0xC0, 0x2B} TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
        {0xC0, 0x2F} TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
        {0x00, 0x9E} TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
        {0xC0, 0x24} TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
        {0xC0, 0x28} TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
```

```
    {0x00, 0x6B} TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
    {0xC0, 0x23} TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
    {0xC0, 0x27} TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
    {0x00, 0x67} TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
    {0xC0, 0x0A} TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
    {0xC0, 0x14} TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
    {0x00, 0x39} TLS_DHE_RSA_WITH_AES_256_CBC_SHA
    {0xC0, 0x09} TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
    {0xC0, 0x13} TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
    {0x00, 0x33} TLS_DHE_RSA_WITH_AES_128_CBC_SHA
    {0x00, 0x9D} TLS_RSA_WITH_AES_256_GCM_SHA384
    {0x00, 0x9C} TLS_RSA_WITH_AES_128_GCM_SHA256
    {0x00, 0x3D} TLS_RSA_WITH_AES_256_CBC_SHA256
    {0x00, 0x3C} TLS_RSA_WITH_AES_128_CBC_SHA256
    {0x00, 0x35} TLS_RSA_WITH_AES_256_CBC_SHA
    {0x00, 0x2F} TLS_RSA_WITH_AES_128_CBC_SHA
    {0x00, 0xFF} TLS_EMPTY_RENEGOTIATION_INFO_SCSV
  compression_methods (len=1)
    No Compression (0x00)
  extensions, length = 169
    extension_type=server_name(0), length=14
      0000 - 00 0c 00 00 09 6c 6f 63-61 6c 68 6f 73 74
.....localhost
    extension_type=ec_point_formats(11), length=4
      uncompressed (0)
      ansiX962_compressed_prime (1)
      ansiX962_compressed_char2 (2)
    extension_type=supported_groups(10), length=4
      ecdh_x25519 (29)
    extension_type=session_ticket(35), length=0
    extension_type=encrypt_then_mac(22), length=0
    extension_type=extended_master_secret(23), length=0
    extension_type=signature_algorithms(13), length=58
      sig_p256 (0x0901)
      sig_p384 (0x0902)
      sig_p521 (0x0903)
      sig_x25519 (0x0904)
      sig_x448 (0x0905)
      ecdsa_secp256r1_sha256 (0x0403)
      ecdsa_secp384r1_sha384 (0x0503)
      ecdsa_secp521r1_sha512 (0x0603)
      ed25519 (0x0807)
      ed448 (0x0808)
      rsa_pss_pss_sha256 (0x0809)
      rsa_pss_pss_sha384 (0x080a)
      rsa_pss_pss_sha512 (0x080b)
      rsa_pss_rsae_sha256 (0x0804)
      rsa_pss_rsae_sha384 (0x0805)
      rsa_pss_rsae_sha512 (0x0806)
      rsa_pkcs1_sha256 (0x0401)
      rsa_pkcs1_sha384 (0x0501)
      rsa_pkcs1_sha512 (0x0601)
      ecdsa_sha224 (0x0303)
      ecdsa_sha1 (0x0203)
```

```
          rsa_pkcs1_sha224 (0x0301)
          rsa_pkcs1_sha1 (0x0201)
          dsa_sha224 (0x0302)
          dsa_sha1 (0x0202)
          dsa_sha256 (0x0402)
          dsa_sha384 (0x0502)
          dsa_sha512 (0x0602)
        extension_type=supported_versions(43), length=9
          TLS 1.3 (draft 26) (32538)
          TLS 1.2 (771)
          TLS 1.1 (770)
          TLS 1.0 (769)
        extension_type=psk_key_exchange_modes(45), length=2
          psk_dhe_ke (1)
        extension_type=key_share(51), length=38
            NamedGroup: ecdh_x25519 (29)
            key_exchange:  (len=32):
    DAE3291AB70469BF32CFB1A30F44282E9CBDCD1B56452437A81CB55BCED69A3F


Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = Handshake (22)
  Length = 122
    ServerHello, Length=118
      server_version=0x303 (TLS 1.2)
      Random:
        gmt_unix_time=0xA60025A5
        random_bytes (len=28):
    DE6ABC4EBF4CB2733B19965F31D420C4C369621C0F0D0390EAB71173
      session_id (len=32):
    FAFE6D982F9FEABAE5767008249547DCECE39BBD58379A58CA6335E8ADA25259

      cipher_suite {0x13, 0x02} TLS_AES_256_GCM_SHA384
      compression_method: No Compression (0x00)
      extensions, length = 46
        extension_type=supported_versions(43), length=2
            TLS 1.3 (draft 26) (32538)
        extension_type=key_share(51), length=36
            NamedGroup: ecdh_x25519 (29)
            key_exchange:  (len=32):
    B939712137A6B8F394649C28B5351F8AC13A80A883EB857866A41416540D814B


Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ChangeCipherSpec (20)
  Length = 1
    change_cipher_spec (1)

Sent Record
Header:
```

```
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 39
  Inner Content Type = Handshake (22)
    EncryptedExtensions, Length=18
      extensions, length = 16
        extension_type=supported_groups(10), length=12
          secp521r1 (P-521) (25)
          ecdh_x25519 (29)
          secp256r1 (P-256) (23)
          ecdh_x448 (30)
          secp384r1 (P-384) (24)

Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 359
  Inner Content Type = Handshake (22)
    Certificate, Length=338
      context (len=0):
      certificate_list, length=334
        ASN.1Cert, length=329
------details-----
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            f5:d1:34:cd:4c:4f:76:7f
        Signature Algorithm: ED25519
        Issuer: C = NL, ST = Some-State, L = Nijmegen, O = RU,
   OU = ICIS
        Validity
            Not Before: Feb 24 22:09:10 2018 GMT
            Not After : Mar 26 22:09:10 2018 GMT
        Subject: C = AU, ST = Some-State, O = Internet Widgits
   Pty Ltd
        Subject Public Key Info:
            Public Key Algorithm: X25519
                X25519 Public-Key:
                pub:
                  7e:44:aa:5f:ce:67:c7:71:27:a7:b6:66:98:55:73:
                  8c:72:ea:4d:f4:a7:23:0b:ab:9e:92:80:32:bb:d9:
                  91:65
    Signature Algorithm: ED25519
         08:5a:9b:bd:a3:04:b3:bd:a4:d7:06:49:6a:5b:33:93:13:8f:
         02:4f:8c:c2:0d:6b:22:72:35:9e:b9:00:29:7b:99:7f:a5:d6:
         4b:e4:d9:94:8e:ee:bd:aa:ac:4a:23:15:e2:bf:19:e5:aa:c2:
         6d:ca:8e:ae:f9:79:cd:39:80:07
-----BEGIN CERTIFICATE-----
MIIBRTCB+AIJAPXRNM1MT3Z/MAUGAytlcDBRMQswCQYDVQQGEwJ
OTDETMBEGA1UECAwKU29tZS1TdGF0ZTERMA8GA1UEBwwITmlqbW
VnZW4xCzAJBgNVBAoMAlJVMQ0wCwYDVQQLDARJQ0lTMB4XDTE4M
DIyNDIyMDkxMFoXDTE4MDMyNjIyMDkxMFowRTELMAkGA1UEBhMC
```

```
QVUxEzARBgNVBAgMClNvbWUtU3RhdGUxITAfBgNVBAoMGEludGV
ybmV0IFdpZGdpdHMgUHR5IEx0ZDAqMAUGAytlbgMhAH5Eql/OZ8
dxJ6e2ZphVc4xy6k30pyMLq56SgDK72ZFlMAUGAytlcANBAAham
72jBLO9pNcGSWpbM5MTjwJPjMINayJyNZ65ACl7mX+l1kvk2ZSO
7r2qrEojFeK/GeWqwm3Kjq75ec05gAc=
-----END CERTIFICATE-----
------------------
        extensions, length = 0

Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 73
  Inner Content Type = Handshake (22)
    CertificateVerify, Length=52
      Signature Algorithm: sig_x25519 (0x0904)
      Signature (len=48): 63
    E2B02740E6ADFC0B7F7857731AC109B9A758E4F38CA837F \
      08D3824E2CE521EB5BDE3D03A678E0650943CA2724FEE5B

Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 69
  Inner Content Type = Handshake (22)
    Finished, Length=48
      verify_data (len=48): 20422
    B535A5C12DDDD21C23DEDB11E1F11526E48AD3CC663B \
      FA71A6A63FDAE9306EECC42BF1AE5707F0F10D7A0972A35

Received Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ChangeCipherSpec (20)
  Length = 1
Received Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 69
  Inner Content Type = Handshake (22)
    Finished, Length=48
      verify_data (len=48): 60
    FF60AB0AF2B90147EDCD8CF776A93A6F24FFBEDFC504CB7 \
      FE07CA84F15983740AF7D35EF4D94050672F028910F026D

Sent Record
Header:
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 259
  Inner Content Type = Handshake (22)
```

```
    NewSessionTicket, Length=238
        ticket_lifetime_hint=7200
        ticket_age_add=2983685265
        ticket_nonce (len=1): 00
        ticket (len=224):
        7BE86DDBB50FAB33CAFECECD81A39ACABB59EF543E278EBA9D7
        DE5B5B7BC9B99722C65451BB8B7EC67C69B14442727488D388B
        840868C49AB89BB7847623FA34D72C59AB017BCBA5F9E3A271F
        96CCB9350BA45F53E887C9E870B1A3F4380AC81EA668A3E63FF
        1191061D84D68C4AD1AACDAD691A6DAB913F9ABA91CF6706599
        66F48CB81D6BCE694607D2107E422EB28F31514064736F06C92
        6763A7FECB813A6A14F7F4FEE376E79618B538187B8A24245C6
        F81637DA491E94D8918A38209A8711187712BFA71C4520F38DA
        3672792EBDAFA751FE62BB10476BD07FCF8E2BBC
        extensions, length = 0

-----BEGIN SSL SESSION PARAMETERS-----
MIGQAgEBAgIDBAQCEwIEIKhOr+yPeonxsrADTeyqgmYK3i6Hu1V
yey8lMB/9CGXSBDA2RB/TU77Wggc7GHIYOw6TvQeHePuQmIhh4w
lMaWWmVIz8fBVLA/xSYYXh41nWSYGhBgIEW3a6laIEAgIcIKQGB
AQBAAApgsECWxvY2FsaG9zdK4HAgUAsddskbEDBAEA
-----END SSL SESSION PARAMETERS-----
Shared ciphers:TLS_AES_256_GCM_SHA384:
    TLS_CHACHA20_POLY1305_SHA256:TLS_AES_128_GCM_SHA256:ECDHE-
    ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA
    -AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-
    CHACHA20-POLY1305:DHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-
    AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-
    AES128-GCM-SHA256:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-
    AES256-SHA384:DHE-RSA-AES256-SHA256:ECDHE-ECDSA-AES128-
    SHA256:ECDHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA256:ECDHE-
    ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES256-SHA:
    ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES128-SHA:DHE-RSA-AES128-
    SHA:AES256-GCM-SHA384:AES128-GCM-SHA256:AES256-SHA256:
    AES128-SHA256:AES256-SHA:AES128-SHA
Signature Algorithms: ECDSA:ECDSA:ECDSA:0x04+0x09:0x05+0x09:
    ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA512:Ed25519:Ed448:RSA-
    PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA512:RSA-PSS+SHA256:RSA
    -PSS+SHA384:RSA-PSS+SHA512:RSA+SHA256:RSA+SHA384:RSA+SHA512
    :ECDSA+SHA224:ECDSA+SHA1:RSA+SHA224:RSA+SHA1:DSA+SHA224:DSA
    +SHA1:DSA+SHA256:DSA+SHA384:DSA+SHA512
Shared Signature Algorithms: ECDSA:ECDSA:ECDSA:0x04+0x09:0x05+0
    x09:ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA512:Ed25519:Ed448:
    RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA512:RSA-PSS+SHA256
    :RSA-PSS+SHA384:RSA-PSS+SHA512:RSA+SHA256:RSA+SHA384:RSA+
    SHA512:ECDSA+SHA224:ECDSA+SHA1:RSA+SHA224:RSA+SHA1
Supported Elliptic Groups: X25519
Shared Elliptic groups: X25519
---
No server certificate CA names sent
CIPHER is TLS_AES_256_GCM_SHA384
Secure Renegotiation IS supported
Received Record
Header:
```

```
  Version = TLS 1.2 (0x303)
  Content Type = ApplicationData (23)
  Length = 19
  Inner Content Type = Alert (21)
    Level=warning(1), description=close notify(0)

DONE
shutting down SSL
CONNECTION CLOSED
ACCEPT
```

Appendix C

Reproducing

To reproduce our benchmarks for either OPTLS or the Internet draft the code can be found at:

- https://github.com/dqi/openssl/tree/OPTLS-InternetDraft
- https://github.com/dqi/openssl/tree/OPTLS

We provide `python` programs with the objective of making it easy to reproduce the benchmarks.

1. Clone the repository.
2. Switch to the right branch:
    - `git checkout OPTLS`
    - `git checkout OPTLS-InternetDraft`
3. `./config enable-ssl-trace enable-ec_nistp_64_gcc_128 -DMYBENCH`
4. `make`
5. To benchmark run `bench_id.py`, `bench_optls.py` or `bench_13.py` in the `benchmarks` directory.

To compile `bench_ed25519.c` and `bench_x25519.c` use the following commands from within the `benchmark` folder:

- `gcc bench_ed25519.c -I../include -L../ -lcrypto -o ed25519`
- `gcc bench_x25519.c -I../include -L../ -lcrypto -o x25519`

Note that it may be necessary to override the default OpenSSL libraries; setting `LD_LIBRARY_PATH` is useful:

- `export LD_LIBRARY_PATH=/path/to/cloned/repo/openssl/`