Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

The transformation of TLS from version 1.2 to 1.3

Efficiency vs Security vs Interoperability

Bachelor's Thesis

Samuel Bedassa Alemu

samuelbe@ethz.ch

Applied Cryptography Group
Institute of Information Security, Department of Computer Science
ETH Zürich

Supervisor:
Prof. Dr. Kenneth Paterson

March 31, 2020

Acknowledgements

I would like to thank my supervisor, Kenny Paterson, for his interest and guidance. The lengthy discussions that have taken place over the course of this thesis have provided me with an invaluable learning experience. I would also like to thank my friends and family, for their constant support, especially my mom, for her frequent prayers.

Abstract

The Transport Layer Security (TLS) protocol is the de-facto means of securing communication on the Internet. The protocol is used by billions of people on a daily basis and is virtually the foundation of our modern Internet-based civilization. Throughout its more than two decades of history, TLS has been the subject of numerous security researches and several high-profile attacks. Recently, the pressure to improve performance, along with the many weaknesses identified, has led the IETF to develop a new version of the protocol, namely TLS 1.3.

In developing the new version of the protocol, the IETF pursued three main groups of objectives: efficiency, security and interoperability. In this thesis, we take a closer look at the IETF's endeavours to meet these objectives. After giving a detailed summary of the protocol, we name the specific objectives, discuss the incentives to pursue them and identify possible design trade-offs between the groups of objectives. We describe the defining design decisions, the circumstances that required them, and their concrete implications in the protocol. Besides, we untangle workarounds needed to make the protocol functional and evaluate their effectiveness. We conclude by commenting on the current priorities of the protocol.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

The transformation of TLS from version 1.2 to 1.3

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Bedassa Alemu	Samuel

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Wallenwil, 31.03.2020	[illegible]

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

Contents

Chapter 1

Introduction

The Internet has become a platform on which much of our everyday life takes place. Many services, including markets, banking and news, are in transition or have already migrated to their "online" version. The gradual increase in the importance of the platform and our growing dependence on it is irrefutable. An essential part of this growth and many innovative applications has been the ability to send information with some security guarantees. In particular, it is crucial that the information transmitted is not manipulated, forged or read by anyone other than the sender and receiver. As applications become increasingly sensitive and consumers become security-conscious, Internet traffic is becoming more and more encrypted. Google has reported that more than 90% of traffic across its services and websites was encrypted in January 2020, up from 50% in January 2014 [1].

For more than two decades, TLS has been the most widely used protocol for secure Internet communication. As a result, it has become a protocol on which our new Internet-based civilization fundamentally depends. Recently, this protocol has been updated to ensure better privacy, security and performance. The new version is poised to provide the foundation for encrypted Internet communications for decades to come. In this thesis, we take a closer look at the transformation of TLS into its new version. In particular, we analyze the design decisions that characterize the protocol and identify trade-offs between its objectives.

Given the vitality and eventual longevity of TLS 1.3, we believe it is warranted that key design decisions are identified, reasoned and evaluated. With this thesis, we attempt to capture the consensus regarding compromises made during the protocol's standardization process. Doing so may facilitate comparisons between the transformation at hand and previous ones or allow predictions about future redesigns. It may also contextualize possible future attacks due to compromises made. Furthermore, differentiating between ordinary protocol changes and incidental workarounds is crucial. Such an analysis helps to clarify the limits in protocol redesign, leads to a better understanding of previous version iterations and illustrates the state of the protocol's ecosystem.

[1] https://transparencyreport.google.com/https/overview

This thesis is structured as follows: In Chapter 2, we give a brief background to TLS. We describe the protocol in general and introduce the actors involved in its development. We also give a quick overview of attacks and their impact on the protocol. In Chapter 3, we provide a detailed summary of TLS 1.3. We highlight the main differences to previous versions and describe the defining new features. In Chapter 4, we outline the objectives of TLS 1.3, describe the various design decisions that were made to achieve them, and point out the trade-offs made when the objectives collided. In Chapter 5, we briefly evaluate the trade-offs between each group of objectives. Chapter 6 concludes this thesis.

Chapter 2
Background

2.1 TLS/SSL and the IETF

Transport Layer Security TLS is the most widely used protocol for encrypted communication on the Internet and is considered the de-facto standard. Netscape Communications developed its predecessor, Secure Sockets Layer SSL, in the 1990s. SSL Version 2 was released in 1995, followed by SSL Version 3 in 1996. The IETF introduced TLS in 1996 with its decision to standardize a version of SSL in response to the growing need for e-commerce support and the increasing adoption of SSL. TLS 1.0 was released in 1999, followed by TLS 1.1 in 2006, TLS 1.2 in 2008, and TLS 1.3 in 2018. At the time of writing, TLS 1.2 is the most site supported version, with 96.5%, TLS 1.1, TLS 1.0, and TLS 1.3 follow with 73%, 62.9%, and 22%, respectively.[1] In the following, we will use TLS when referring to both TLS and SSL in general.

The primary purpose of TLS is to establish a secure channel between the communicating entities, namely the client and the server [Res18, Section 1.]. This secure channel provides confidentiality and integrity of data during transmission over untrusted networks. In particular, the protocol provides these security services for protocols that run at the application layer. TLS is used in HTTPS for encrypted web browsing, combined with IMAP or SMTP for cryptographically protected e-mail traffic and also for secure communication with embedded systems, mobile devices, and payment systems.

The protocol is composed of a number of sub-protocols, the two major sub-protocols being the channel setup protocol called Handshake, and the transport protocol called Record [Res18, Section 2.]. The handshake protocol is used to negotiate all cryptographic parameters. These include the TLS version, the authentication method, the key exchange method and the symmetric key algorithm to be used in the record protocol. The client lists its capabilities in the `ClientHello` message, and the server selects the parameters to be used from those offered in the `ServerHello` message. In the record protocol, symmetric-key cryptogra-

[1]https://www.ssllabs.com/ssl-pulse/

phy is used in combination with a session key and sequence number to protect the application data. For example, if the server and the client agree on the TLS_ECDHE_ RSA_WITH_AES_128_GCM_SHA256 cipher suite in TLS 1.2, RSA is used to authenticate the server with a certificate, ECDHE for key exchange, AES in GCM mode to encrypt the application data, and SHA256 for message authentication [DR08, Appendix A.5.]. A similar protocol called DTLS is also available, which is based on UDP instead of TCP with similar objectives.

Since TLS 1.0, the Internet Engineering Task Force (IETF) has been responsible for maintaining and improving the protocol [PvdM16]. The IETF is a self-organized group of software developers, implementors, and vendors focused on creating and maintaining standards for the Internet. The standardization process is carried out by working groups (WGs) organized in different areas. The WGs publish standards as Request for Comment documents (RFCs) free of charge. Various RFCs define additional specifications, such as cryptographic algorithms and extensions. Besides TLS, the IETF is also responsible for IP, HTTP, and other widely used protocols. This makes the IETF the de-facto technical forum for Internet protocol standards. Later in chapter 4, we will take an in-depth look at the various decisions the IETF had to make in the standardization process of TLS 1.3.

Based on the standards of the IETF, the vendors implement the actual software to be used. Often these implementations have subtle differences, making some implementations more vulnerable to attacks than the others [BBD$^+$17]. Popular server-side implementations include OpenSSL, GnuTLS and NSS Secure Transport. On the client-side, most connections come from popular browsers like Firefox, Chrome, and Safari. Due to this somehow decentralized nature of the TLS ecosystem, the state of TLS is dependent on many factors, and changes need the coordination of several actors.

2.2 Attacks

Several attacks and security issues have plagued TLS in its more than two decades of history. Consequently, an iteration of changes in the server/client software and in the protocol itself was necessary. From the early days of the Bleichenbacher attack on RSA in SSLv3 to the Selfie attack on TLS 1.3, this widely used protocol has experienced more than two dozen high-profile attacks. The circumstances that made these attacks possible, their complexity and practical feasibility, the theoretical foundations exploited, and the countermeasures adopted were very diverse. Often these attacks were implementation-dependent since different implementations interpret the RFCs in different ways. There have been compression attacks such as CRIME, TIME, and BREACH, downgrade attacks such as FREAK and LOGJAM, and attacks on CBC mode encryption such as BEAST, Lucky 13, and POODLE, to name a few. In the following, we give a brief glimpse into TLS attacks with sample attacks. We will explain the security flaws that made them possible and what impact they had on the protocol.

Lucky 13

In 2013, Alfardan et al. [AP13] published a padding oracle attack on CBC mode block cipher encryption, which was the only option besides RC4 stream cipher encryption in TLS 1.1 and earlier. The attack was based on a padding oracle, which tells an attacker whether or not padding was correctly formatted, which was shown to exist in TLS in earlier works [Vau02]. In particular, the record processing time during decryption needed to be the same regardless of whether the padding was correct or not. If this was not the case, a time difference between good and bad padding would be apparent in the time it takes for error messages to appear on the network. The padding oracle with a delicate timing analysis would provide a decryption capability. Lucky 13 presented a set of attacks, including a full plain text recovery attack. The attacks were also made possible by a coincidence of various factors such as the MAC tag size, the block size of block ciphers, and the number of header bytes. The attacks required the analysis of a large amount of ciphertext and were well suited for data transmitted in many sessions at the same position, such as passwords and HTTP cookies.

Attacks on RC4

RC4 is a stream cipher that has been commonly used since 1994 as part of a variety of cryptosystems, including TLS. The stream cipher has been particularly popular for its short description and fast software implementation. However, it is also known to have several weaknesses. Shortly after Lucky 13, another set of attacks by Alfardan et al. [ABP+13] was made public, which exploited these weaknesses to recover plaintexts. The attacks were based on biases in the RC4 keystream, i.e. deviations from the uniform distribution at certain positions. One of the attacks exploited a complete view of all single-byte biases that occur in the first 256 bytes of the keystream. These biases had long been known at the time, and it had been recommended to routinely dropped the first few hundred keystream bytes before encryption. Nevertheless, popular implementations such as OpenSSL were using RC4 encryption without taking the recommendation into account. The publishers of the attack admitted that the attack was not sophisticated and stated that it was alarming that such a simple attack on TLS was possible. With additional attacks by Vanhoef et al. [VP16] the use of RC4 in TLS was made indefensible.

FREAK

As Beurdouche et al. [BBD+17] argue, TLS has been popular in part because of its flexibility. The protocol supports a variety of authentication modes, key exchange methods, extensions, and protocol versions. For example, TLS 1.2 specifies 37 cipher suites [DR08, Appendix A.5.], which add to the many cipher suites already specified in previous versions. Some of the early cipher suites are

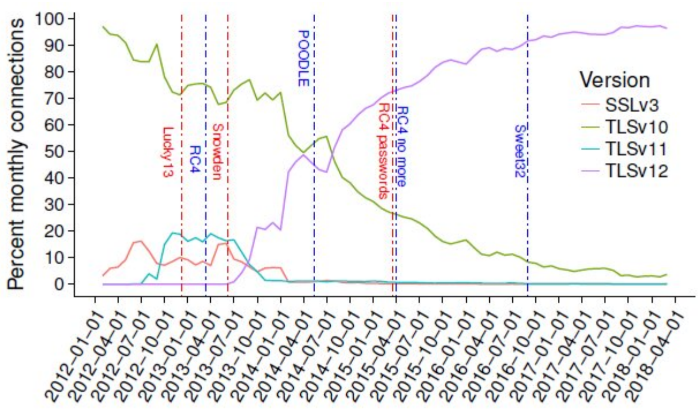


Figure 2.1: Negotiated TLS Versions and high-profile attacks[KRA+18]

known to be unsafe but are still supported for backward compatibility. For example, Export RSA, the legacy cipher suite used to mount the FREAK attack, has been intentionally weakened to comply with U.S. export regulations of the 1990s and was included in SSLv3 and TLS 1.0. Although these regulations have since been relaxed, many libraries contain code to handle these cipher suites, and some servers still support them. This legacy issues, coupled with the complexity of implementing the combination of the wide range of options, presented a security vulnerability. Factoring RSA EXPORT keys, dubbed FREAK [BBD+17], is a downgrade attack made public in 2017 that forces the client and server to use RSA keys with a lower security level. The publishers of the attack have shown that the sub-strength keys of RSA export can be factored within hours in today's hardware. Since many implementations reuse the same public key, a factorized key could be used for an attack lasting for days. The attacker would then be able to read what the client sends to the server and send messages to the client as the server. Several patched implementations have been released in response to this attack.

TLS attacks, including those described above, have played an integral role in the adoption and demotion of TLS versions and certain cipher suites. For instance, SSLv2 and SSLv3 have become deprecated in 2011 and 2015, meaning that TLS servers and clients are required not to negotiate these versions when establishing connections as they no longer provide sufficient security considering many attacks [TP11] [BTPL15]. Lucky 13 and the attacks on RC4 have arguably played a major role in the faster adoption of Authenticated Encryption with Associated Data (AEAD), thus TLS 1.2. Notably, until the introduction of AEAD in TLS 1.2, the cipher suites in TLS were "MAC-then-Encrypt", using either block ciphers in CBC mode or RC4 stream cipher for encryption. A sequence of attacks on CBC mode encryption, namely BEAST (2011), Lucky 13 (2013), POODLE (2014), and attacks on RC4 from 2013 onwards, forced a shift away from these cipher suites and thus a shift to the then-latest TLS 1.2. The recommended countermeasures were in case of POODLE completely disabling SSLv3, in case of BEAST switching to TLS 1.1 or 1.2, in case Lucky 13 and RC4 attacks switching to TLS 1.2. Kotzias et al. [KRA+18] showed in their study of TLS connections that the use of RC4 and CBC mode encryption significantly decreased in 2018 compared to their predominant use in 2012 primarily in response to these attacks. In particular, RC4 had disappeared entirely in the 2018 measurements, and the use of CBC mode had dropped to 10%.

Chapter 3

TLS 1.3

In the spring of 2014, the IETF began drafting TLS 1.3 in response to the many weaknesses identified in TLS 1.2 and growing demand for better performance. After 28 drafts and more than four years of a standardization process, TLS 1.3 was released in August 2018. The new version of TLS brought enhanced security by using modern cryptographic algorithms, improved performance by introducing a new protocol flow, and achieved better privacy by encrypting more of the protocol. In the later part of this chapter, we describe the protocol as required for the subsequent analysis, according to the RFC [Res18].

The standardization process of TLS 1.3 differed from previous versions in its proactive fashion, and its deployment was notably rapid. Unlike the standardization process of earlier versions, there were several contributions from the research community. Besides, several tools were used for the protocol analysis to ensure security objectives. This process was a departure from the IETF's earlier approach of design-release-break-patch to a new design-break-fix -release standardization process, as described by Paterson et al. [PvdM16]. The deployment pace of TLS 1.3 also differed from previous versions, especially that of TLS 1.2. As Holz et al. showed [HARV19], TLS 1.2 connections accounted for essentially 0% of TLS connections in 2012, four years after its release. In April 2019, just a few months after its release, TLS 1.3 accounted for 4.6% of TLS connections. Faster deployment was mainly due to major Internet companies such as Cloudflare, Google, Facebook, and Mozilla. In particular, their control over both endpoints of the connection has made faster deployment easy. It has also enabled them to experiment with the protocol prior to its release.

3.1 Major differences to TLS 1.2

TLS 1.3 has a significant structural departure from previous versions, even though it was initially intended to be an incremental improvement to TLS 1.2 [PvdM16]. Radical changes were necessary to achieve the desired efficiency and not to carry the shortcomings of older versions. In the following, we will list some significant differences to previous versions, mainly to TLS 1.2.

The handshake has been restructured to reduce the number of necessary round trips from two to one [Res18, Section 1.2.]. Encrypted application data can now be sent after only one round trip. Besides, a zero round-trip time (0-RTT) mode has been introduced, which saves a round trip for some application data, at the cost of some security properties. 0-RTT is achieved by using a Pre-Shared Key (PSK), described in the following sections. Furthermore, the payload is now encrypted as early as possible to support a higher level of privacy. Any message after the `ServerHello` is now protected. With the introduction of a new `EncryptedExtensions` message, server certificates, and various extensions that were sent unprotected in the `ServerHello` in previous versions can now be sent encrypted. Unnecessary messages like `ChangeCipherSpec` have also been removed to simplify the protocol. Further, the version negotiation mechanism in TLS 1.2 has been deprecated in favour of a supported version list in an extension.

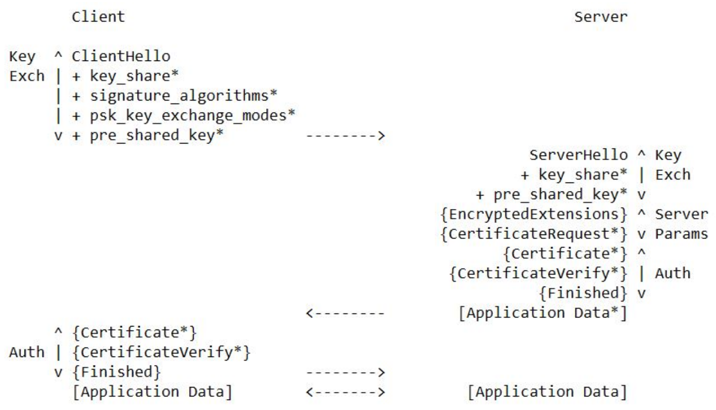
TLS 1.3 supports more secure cryptographic primitives and removes all legacy symmetric encryption algorithms to enhance security. It has only five cipher suites, compared to the several hundred that were allowed in previous versions [Res18, Appendix B.4.]. The five cipher suites are all Authenticated Encryption with Associated Data (AEAD). Furthermore, the concept of cipher suites has changed. A cipher suite in TLS 1.3 is a pair of AEAD and hash algorithms. In particular, key exchange and authentication mechanisms are now negotiated independently.

A single PSK key exchange, to be described in the following section, has replaced both session resumption and PSK-based cipher suites of previous versions [Res18, Section 1.2.]. Moreover, (EC)DHE key exchange is now the only public key-based key exchange mechanism available, which notably provides forward secrecy. For (EC)DHE key exchange, there is a reduced option of five elliptic curve groups (ECDHE) and five finite groups (DHE) [Res18, Section 4.2.7.].

Another change is the redesign of the key derivation functions to allow more straightforward cryptographic analysis [Res18, Section 7.1.]. The new key derivation uses HKDF (HMAC-based extraction and extension key derivation) as an underlying primitive. Significant removals include static RSA and Diffie-Hellman cipher suites, non-AEAD symmetric encryption, i.e. RC4 and CBC mode, custom Diffie-Hellman ephemeral groups (DHE), signature algorithms DSA, MD5, SHA-1 and compression.

3.2 Handshake protocol

The handshake sub-protocol is used at the beginning of each connection to negotiate a protocol version, select cryptographic parameters, and authenticate each other [Res18, Section 4.]. In TLS 1.3, one can think of the handshake as having three phases, as shown in figure 3.1. The key exchange phase is used to



```
         Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^ Server
                                        {CertificateRequest*}  v Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->   [Application Data]
```

Figure 3.1: TLS 1.3 handshake
+ Indicates noteworthy extensions sent in the previously noted message, * Indicates optional or situation-dependent messages/extensions, {} Indicates messages protected using handshake keys, [] Indicates messages protected using application keys

determine the security capabilities of the communicating entities and create a shared keying material, using one of the three key exchange modes. After that, everything is encrypted using a key derived from the negotiated keying material. In the second phase, the server sends parameters that ware not required in the first phase. These include most extensions sent within the `EncryptedExtensions` message and a `CertificateRequest` message that indicates whether client authentication is desired. In the last phase, the server (and optionally the client) authenticates its self with a certificate, additionally with the `Finished` message, a hash of the handshake script, the peers provide key confirmation and the integrity of the handshake.

ClientHello

When a client connects to a server, the first message it sends is a `ClientHello` [Res18, Section 4.1.2]. The structure of this message is the same as in previous versions for interoperability reasons. The message consists of four fields: "legacy_version", "random" (32 bytes nonce), "legacy_session_id", "cipher_suites", "legacy_compression_methods" and "extensions". The legacy fields have no actual relevance, and most parameters are sent via extensions.

With the `ClientHello`, the client provides four sets of options for a cryptographic negotiation [Res18, Section 4.1.1]. First, a list of the cipher suites it supports in the "cipher_suites" field. Second, a list of (EC)DHE groups it supports in "supported_groups" extension, and shares for some or all of these groups in "key_share" extension. Third, a list of PSK identifiers known to the client in a "pre_shared_key" extension, and modes that may be used with the PSKs provided in a "psk_key_exchange_modes" extension. Fourth, signature algorithms the client accepts in a "signature_algorithms" extension and algorithms specific to certificates in a "signature_algorithms_cert" extension.

Depending on the key exchange mode, the client is attempting, one or both of the second and third option sets may be required [Res18, Section 4.2.9.]. Due to a version intolerance problem, the client specifies its version preference with the "supported_versions" extension. This extension must always be present. Otherwise, the message will not be interpreted as a TLS 1.3 `ClientHello`.

ServerHello

A `ServerHello` message is sent in response to a `ClientHello` if the server can find acceptable connection parameters [Res18, Section 4.1.3.]. The structure of this message is the same as in previous versions, as with the `ClientHello`. The message consists of four fields: "legacy_version", "random" (32 bytes nonce), "legacy_session_id_echo", "cipher_suite", "legacy_compression_method", and "extensions". As in `ClientHello`, the legacy fields have no actual relevance.

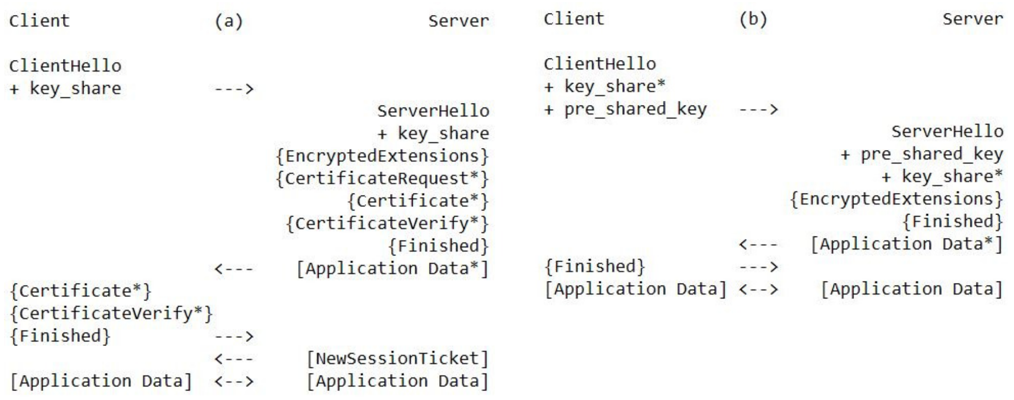
The negotiation on the server-side is done by selecting parameters from the ones provided by the client as follows [Res18, Section 4.1.1]. First, the server selects a single cipher suite to be sent in the "cipher_suite" field. Second, the server selects a single group for which the client has also provided a share. A share corresponding to the selected group is to be sent in a "key_share" extension. Third, the server selects a single PSK key exchange mode and a single PSK identifier to be sent in a "pre_shared_key" extension.

Depending on whether the server has agreed to a PSK key exchange and, if so, which mode it has selected, one or both of the second and third selections may be required [Res18, Section 4.2.9.]. The `ServerHello` must contain a "supported_versions" extension to indicate that it is a TLS 1.3 `ServerHello`, and also to indicate which version is selected and in use. After an exchange of `CleintHello` and `ServerHello` messages, the peers have a shared secret keying material. Extensions other than those already mentioned are sent separately in the `EncryptedExtensions` message, followed by other handshake messages encrypted with a handshake key derived from the established keying material.

PSK and key exchange modes

Once a handshake is complete, the server may send a `NewSessionTicket` message [Res18, Section 4.6.1.]. This message creates a unique association between a label and a secret PSK derived from secrets established in the initial handshake. `NewSessionTicket` includes a PSK identifier or label in the field "ticket" and a value unique across all tickets issued on this connection in the field "ticket_nonce". The PSK associated with the "ticket" field value is generated using HKDF functions as follows.

$$HKDF - Expand - Label(resumption\_master\_secret, "resumption", \\ ticket\_nonce, Hash.length)$$



```
Client                (a)                Server

ClientHello
+ key_share         --->
                                     ServerHello
                                     + key_share
                           {EncryptedExtensions}
                           {CertificateRequest*}
                                  {Certificate*}
                            {CertificateVerify*}
                                      {Finished}
                    <---      [Application Data*]
{Certificate*}
{CertificateVerify*}
{Finished}          --->
                    <---      [NewSessionTicket]
[Application Data]  <-->      [Application Data]
```

```
Client                (b)                Server

ClientHello
+ key_share*
+ pre_shared_key    --->
                                     ServerHello
                                + pre_shared_key
                                    + key_share*
                           {EncryptedExtensions}
                                      {Finished}
                    <---     [Application Data*]
{Finished}          --->
[Application Data]  <-->      [Application Data]
```

Figure 3.2: Handshake with PSK (a) Initial Handshake (b) Subsequent Handshake

Upon receiving a `NewSessionTicket` message, the client can independently compute the PSK using the "ticket_nonce" value and shared secrets established in the initial connection. If it intends to use the PSK in later connections, it sends the corresponding "ticket" value in a "pre_shared_key" extension to the server [Res18, Section 4.2.11.]. The client must also send a "PSK_key_exchange_modes" extension, indicating which modes it supports with the PSKs it sends.

Three Key Exchange Modes in total, including two PSK modes, are defined [Res18, Section 4.2.9.]. With (EC)DHE mode, the connection is established using (EC)DHE shares, as shown in figure 3.2 a. With PSK-only mode, PSK is used to bootstrap the cryptographic state. Additionally, a full handshake is avoided. In particular, peer authentication with a certificate is skipped as authentication happens as a side effect of key exchange. PSK with (EC)DHE mode combines the first two modes, i.e. peers exchange (EC)DHE group shares in addition to PSKs, as shown in figure 3.2 b. PSK with (EC)DHE provides forward secrecy (section 4.2), which PSK-only does not provide. PSKs can also be established out of bound, i.e., without an initial connection.

HelloRetryRequest and downgrade protection

If the server can find acceptable parameters, but the `ClientHello` does not contain enough information to proceed with the handshake the server sends a `HelloRetryRequest` message [Res18, Section 4.1.4.]. This is especially the case if the client has provided a list of supported groups that includes a group that the server also supports, but the client has provided no share corresponding to the commonly supported group.

For interoperability reasons, the `HelloRetryRequest` message has the same structure as `ServerHello`. A client identifies a `ServerHello` as a `HelloRetryRequest` by checking whether the "random" field value of the message matches SHA-256 of the string "HelloRetryRequest". All other fields have the same meaning as in a `ServerHello`, and the message contains a minimal set of extensions that the client needs to generate a correct `ClientHello` for a subsequent connection attempt. In particular, the message contains a "key_share" extension that indicates a mutually supported group that the server intends to negotiate and for which it requests a share [Res18, Section 4.2.8.]. The server may also use a "cookie" extension to offload state to the client.

The "random" value in `ServerHello` also serves a downgrade protection mechanism. When a server running TLS 1.3 negotiates a version below 1.3, it sets the last eight bytes of the "random" field to a unique value specific to the negotiated lower version. A client running TLS 1.3 checks the "random" field value to see if it has negotiated a lower version with a server that is also running version 1.3. If this is the case, the client aborts the handshake. This provides limited protection against a downgrade attack.

3.3 Record protocol

The TLS record protocol takes the data to be transmitted, fragments it into manageable blocks, protects the records, and transmits the result [Res18, Section 5.]. Received data is verified, decrypted, reassembled, and then delivered to the higher-level clients.

Three structures are used for this process: TLSPlaintext, TLSInnerPlaintext, and TLSCiphertext. The first one contains the data to be transmitted with a maximum length of 2^{14} bytes. The second contains the data fragment in the first structure padded with zeros, separated by a "type" field, with values such as "application_data", "handshake" and "alert". The third structure contains the encryption of the second structure, using record protection functions. The AEAD cipher suites used in these protection functions notably provide a unified encryption and authentication operation [Res18, Section 5.2.]. When a TLSCiphertext is received, decryption functions reverse the process to obtain a TLSInnerPlaintext.

As with the handshake protocol, the record protocol takes interoperability into account. For example, a TLSPlaintext of the type "change_cipher_spec" may appear unencrypted at any time after a `ClientHello` and before the peer's `Finished` message. This recording is to be dropped on arrival. Further, TLSCiphertext has an "opaque_type" field fixed to the value "application_data" for middlebox compatibility. The actual type of the transmitted data is to be found in TLSInnerPlaintext after decryption. TLSCiphertext also has a "legacy_record _version" field to indicate the TLS version in use, which is redundant since encrypted records are not sent until a version has been negotiated.

0-RTT

When a client attempts to establish a connection with PSK, it can send application data immediately after the `ClientHello` message [Res18, Section 2.3.]. The transmitted data is encrypted with a key derived from the first PSK indicated in the "pre_shared_key" extension in the `ClientHello`. The client also sends an "early_data" extension to indicate the use of this functionality [Res18, Section 4.2.10.]. The handshake proceeds as with any other handshake with PSK, as shown in Figure 3.2 b. Additionally, the server sends the extension "early_data" in its `EncryptedExtensions` message to indicate that it allows early data to be transmitted with its PSKs and is processing the early data sent by the client if any. After the client receives the `Finished` message from the server, it sends the `EndOfEarlyData` message to indicate the end of the 0-RTT data transmission and the change of key [Res18, Section 4.5.].

0-RTT is an additional advantage of the PSK key exchange modes in addition to avoiding computationally expensive public-key authentication. The parameters of the transmitted early data are those associated with the PSK used. In particular, for PSKs established via a `NewSessionTicket` message, those negotiated in the connection that established the PSK. The security properties of early data are weaker than those of other types of TLS data. Namely, it does not provide forward secrecy and non-replay guarantees (section 4.2).

3.4 Extensions

Since the structure of the `ClientHello` and `ServerHello` messages remains unchanged for interoperability reasons, functionality has been moved to extensions [Res18, Section 4.1.2.]. TLS 1.3 defines a total of twenty-two extensions, of which twenty-one may appear in `ClientHello`, three in `ServerHello`, and nine in `EncryptedExtensions` [Res18, Section 4.2]. Extensions may also appear in other handshake and post-handshake messages.

Extensions are generally structured in a request/response fashion, although some extensions are only indications without a corresponding response. The client sends its extension requests in `ClientHello`, and the server sends its extension responses in `ServerHello`, `EncryptedExtensions`, `HelloRetryRequest` and `Certificate`. Similarly, the server sends extension requests in `CertificateRequest`, and the client responds in `Certificate`. The server can also send unsolicited extensions in `NewSessionTicket`, though the client does not respond directly to these.

In TLS 1.3, the use of certain extensions is mandatory, as described in the sections above. Furthermore, unlike in TLS 1.2, extensions are negotiated for each handshake, even when using PSK key exchange modes. However, the 0-RTT parameters are those negotiated in the previous handshake.

Chapter 4

Parameters of a trade-off in the design of TLS 1.3

In the mid-1990s, when the earliest versions of TLS were developed, the protocol arguably had only one goal. As its name suggests, to provide secure communication over the Internet. At that time, the use of the protocol was limited to e-commerce. In particular, such applications relied only on the functionality of the protocol. In the mid-2010s, when TLS 1.3 was developed, the landscape was radically different. On the one hand, the popularity and ubiquitous use of the protocol meant that it no longer had only one goal. Applications of TLS have expanded from the initial use of e-commerce to modern applications such as streaming and IoT. In particular, efficiency was no longer a mere side matter Decades of protocol adoption also meant that a cluster of infrastructure was built around the protocol. Unfortunately, these infrastructures are often faulty and seldom change. The situation has gone so far that the infrastructures severely limit the protocols' modification. In order to be functional, each new version of TLS must take several aspects of interoperability into account. In this chapter, we group the goals of TLS 1.3 into three, describe the challenge in achieving them, identify trade-offs between the set of goals, and describe the consequences of such trade-offs for the protocol.

4.1 Efficiency

TLS 1.3 stands out for its efficiency goals more than any other, as reflected in its new and central features. Namely 1-RTT handshake, early data transmission and PSK key exchange modes. The efficiency goals could be divided into two groups: Reducing latency and reducing computation. In the following, we describe the efficiency goals and name the most important incentives for achieving them.

4.1.1 The challenge from QUIC

Google's Quick UDP Internet Connections (QUIC) is an application layer transport protocol that relies on UDP transport. QUIC is designed to provide high performance, reliable in-order packet delivery and security guarantees similar to TLS [KJC+19]. Google was compelled to start developing the protocol in 2012 due to performance issues encountered with HTTP when running over TCP/TLS. QUIC was initially released in 2013 and has since undergone rapid development. Recently, a working group was formed at the IETF to promote it as a standard for web content delivery [CMTH17].

QUIC has significantly benefited from the continuous improvements made since its release. By tightly coupling the concepts of TCP, TLS and HTTP, the protocol is able to use cross-layer information and evolve much faster than its counterparts [WRWH19] [CJJ+19]. A significant advantage of QUIC over TLS has been the fact that it only requires 1-RTT to establish a connection. TLS 1.3, with its new protocol flow, achieves the same latency when only considering the handshake. Since QUIC runs over UDP instead of TCP, it still retains a 1-RTT advantage when considering the broader network stack. Notably, TLS requires an initial TCP handshake, which also takes 1-RTT. QUIC also provides a 0-RTT feature that allows a client to start a new session without a full handshake. For this, a client must have previously communicated with the server and a limited state is stored. This has been another advantage over previous versions of TLS. As we describe later, TLS 1.3 makes several security compromises to achieve the same capability.

Studies have shown that QUIC outperforms HTTP + TLS + TCP in almost every scenario [KJC+19] [CJJ+19] [LRW+17] . In particular, QUIC achieves a better quality of experience for video streaming and when used in unstable networks, such as wireless mobile networks. In comparisons made before the release of TLS 1.3, QUIC reduces Google Search latency by 8% for desktop and 3.5 % for mobile users. Furthermore, QUIC reduces video buffer time by 18% for desktop and 15.3% for mobile users. Currently, the use of QUIC is limited to Google servers and the Chrome/Chromium browser. Nevertheless, these already represent a considerable part of the network traffic. As of 2016, more than 85% of Chrome requests to Google servers use QUIC [CMTH17]. The performance advantages of the protocol, Google's push and the IETF's standardization process make QUIC a significant challenger to TLS or even a replacement.

4.1.2 Age of IoT

The Internet of Things (IoT) is a system of interconnected computing devices. These include smart homes, health care devices, and autonomous driving systems. These devices are capable of transmitting data over a network without the need for human-to-human or human-to-computer interaction. Because of their

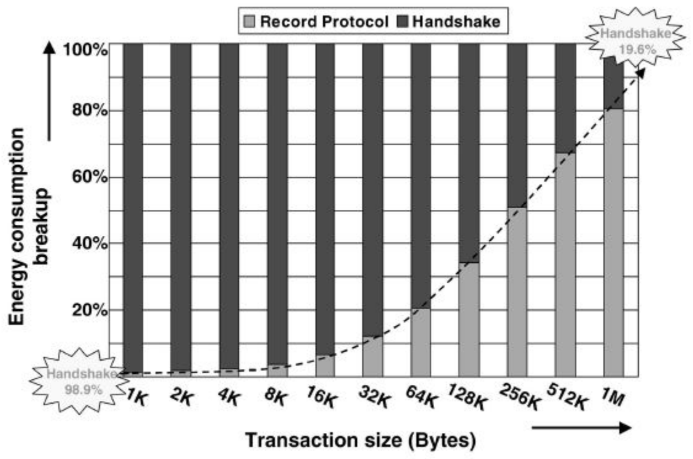


Figure 4.1: Variation of energy consumption with increasing transaction sizes [PRRJ06]

widespread use, some studies predict that there will be nearly 50 billion smart, interconnected objects by the end of 2020 [BDK16]. Like any device connecting over the Internet, IoT devices need security. TLS has been proposed to achieve this goal. Unfortunately, the high security of TLS comes at the cost of high computational and energy demand. Primarily, this is due to the cryptographic algorithms incorporated in the protocol. The limited resources available in IoT devices pose the challenge to achieve a certain level of security while minimizing resource consumption.

The goal of energy-efficient execution of security protocols can be achieved in several ways, which can be divided into two broad classes [PRRJ06]. First, by making the execution of cryptographic primitives efficient through a combination of hardware and software techniques. Secondly, by allowing the protocols to change their mode of operation depending on the operating environment. The second class may involve a deliberate compromise between security and energy consumption. The trade-offs may come in the form of adjusting the key length, giving preference to energy-efficient cipher suites, or avoiding computationally intensive steps. In the case of TLS, public key authentication is most critical, as it is a significant part of the overall energy dissipation. Especially for small data transactions, the energy consumed by cryptographic processing consists almost exclusively of asymmetric algorithm contributions. As shown in figure 4.1, with increasing transaction size, symmetric algorithms replace asymmetric algorithms as the dominant contributor. For example, in a transaction with 1 KB of data, 58% of the energy is consumed by cryptographic algorithms. In such a transaction, asymmetric algorithms account for more than 90 % of the energy consumption of cryptographic processing and about 56 % of the total energy consumption [PRRJ06].

The design of TLS 1.3 tries to strike a balance between security and resource consumption, partly in view of IoT applications. For devices with short and frequent connections, the PSK key exchange modes would make a big difference in avoiding costly computations. In particular, out-of-band PSKs could be hard-coded on such devices, so that the ability to make an initial connection is not even required. Naturally, all this is at the expense of certain security features, as we will describe in the next section.

4.2 Security

In the following, we discuss the security vulnerabilities of previous versions of TLS and their causes, list the security requirements set out for the new version and describe published works proving these requirements. Finally, we note security vulnerabilities in TLS 1.3, especially in PSK and early data.

4.2.1 Vulnerabilities of earlier TLS versions

Over the years, several security vulnerabilities have been discovered in TLS, and an iteration of security enhancements has followed. These weaknesses were diverse in their cause, subtlety and impact. Properly addressing them has accordingly been a challenge. The design of TLS 1.3 attempts to address this challenge and also not to repeat the shortcomings of previous countermeasures by adopting new approaches. In the following, we group the vulnerabilities of earlier versions of TLS by their cause, describe the countermeasures taken in the past and show what was done differently in the design of TLS 1.3.

Lack of explicit and complete security goals

Several vulnerabilities in previous TLS versions could be attributed to the lack of clear security objectives and the subsequent lack of proof-based protocol engineering. The precise identification of many critical security properties only happened late in the protocol's version iteration [PvdM16]. The handshake protocol, in particular, has suffered severely, consequently. Cipher suite and version rollback, confusion in the key exchange algorithm and in session resumption, to name a few. Most security enhancements were a reaction to publicly disclosed attacks rather than to a predefined goal. Such an approach means that an enhancement only targets a specific case of a vulnerability, i.e. an attack, and not the underlying broader problem. This ad-hoc approach was reflected in the decision to introduce a `Finished` message to combat a man in the middle attack, which alters the cipher suites offered by the client [MS13]. However, the measure left out the `ChangeCipherSpec` message in the added handshake authentication. A further enhancement was necessary that required the `ChangeCipherSpec` message to be received before the `Finished` message to proceed with the connection establishment. Granted, the concept of formal security requirements and verification tools was in its infancy as TLS came to life, determining how its absence has affected the protocol is still vital. Moreover, it explains the importance of the security proofs adopted in the design of TLS 1.3, as well as the challenge resulting from limitations in the still-developing area of proof-based protocol design.

Broken or unsafe cryptographic building blocks

The security gaps in the record protocol have mainly been a result of the use of unsafe or broken cryptographic components. Weaknesses in CBC encryption, information leakage due to compression, and predictable initialization vectors (IVs), to name a few. Notably, most of the high-profile attacks on TLS have exploited these vulnerabilities (section 2.2). Adoption of newer and more secure primitives by the ecosystems has been, as with the above described set of vulnerabilities, a reaction to attacks [KRA+18]. A more conservative approach to cryptographic primitives would have been necessary. Since the biases of RC4 and the padding oracle of CBC mode had been known for a long time, a switch to AEAD as soon as TLS 1.2 was introduced would have been appropriate. As the main objective of the protocol is secure communication, the IETF should have deprecated unsafe cipher suites and versions sooner. Notably, the security vulnerabilities caused by broken or unsafe primitives are the result of a disconnect between academics and practitioners [BL16]. The lack of response to theoretical weaknesses in cryptography and the need for TLS-specific attacks for actual change in the ecosystem are good illustrations. In contrast, the approach taken in the design of TLS 1.3 to remove any cipher suite considered unsafe is radical.

Flaws by vendors and PKI

In addition to IETF' s design decisions, vendors and public key infrastructure (PKI) play a critical role in the security of TLS. A flaw in the implementation, improper distribution of certificates, or the compromise of a Certificate Authority (CA) have been common in the TLS ecosystem. On the vendor side, the problems were caused by the complex variety of protocol versions, extensions, authentication modes, and key exchange methods. Implementation bugs have allowed several attacks, such as LOGJAM and SKIP [BBD+17]. Formal verification tools are being used to reduce the impact of such problems. On the PKI side, any Certificate Authority can issue a certificate for any domain [SGN+18]. The reliance on a certificate for server authentication makes TLS vulnerable to phishing attacks. Certificate transparency logs have been proposed as a solution and are becoming standard in major browsers.

4.2.2 Requirements of TLS 1.3

In February 2016, two years after the start of the standardization process, it was pointed out that TLS 1.3 was missing a set of complete and explicit requirements [PvdM16]. This was surprising, as the standardization processes had already progressed far without clear security objectives. When these were later added, the IETF had to rely on external formal definitions and proofs to assess the security provided by the protocol [Res18, Appendix E]. The security requirements the protocol aims to achieve are listed as follows.

Handshake protocol

- The peers should have the same session keys after the handshake.
- The session keys should only be known to the communicating peers.
- If a client believes it is communicating with a certain server, the client should indeed be communicating with that server. The analogue should hold for the server in case of mutual authentication.
- Different handshakes should produce different and unrelated session keys.
- The cryptographic parameters should be the same on both sides and should be the same as if the communication was in the absence of an attack.
- If either party's long-term key becomes compromised, sessions completed before the compromise should remain secure. (Forward secrecy)
- Should an attacker compromise the long-term key of one party, the attacker should not be able to use this key to impersonate an uncompromised party in communication with that party.
- The server's identity should be protected against passive attackers. The client's identity should be protected against passive and active attackers.

Record protocol

- An attacker should not be able to determine the plaintext of a given record.
- An attacker should not be able to produce an acceptable new record.
- An attacker should not be able to cause the receiver to accept a record which it has already accepted or cause the receiver to accept record N+1 without having first processed record N. (Non-replayability)
- The attacker should not be able to determine the length of content versus padding given a record.
- An attacker that compromises an endpoint should not be able to decrypt traffic encrypted with an old key. (Forward secrecy)

Contributions from the academic community have enabled a standardization process with proven security properties. For instance, the symbolic model for handshake interaction using the tamarin prover by Cremers et al. [CHH+17] proved the majority of the security requirements mentioned above. Their model was notably able to incorporate the PSK key exchange modes and early data. Due to limitations in the model, properties, such as downgrade protection and endpoint identity protection, were not covered. One flaw discovered in this analysis was that the protocol allows for "silent" certificate rejection. Meaning that the client does not receive explicit confirmation whether the server has successfully received the client's response to a certificate request. In particular, peers may not necessarily share a common view of the session. This is the case with both regular handshake authentication and post-handshake authentication. The TLS working group has decided not to fix this behaviour. If a client wants to be informed about its authentication status, this can be done in the application layer. However, applications may incorrectly assume that sending a client certificate and receiving further messages from the server indicates that the server considers the connection to be mutually authenticated. This may cause problems in the application layer.

4.2.3 PSK and early data

An only PSK key exchange mode notably does not provide forward secrecy. An attacker that learns a traffic secret can compute all future traffic secrets on that connection, including the secret PSKs [Res18, Appendix E.1]. In addition, TLS does not provide security for data transmitted on a connection after the traffic secret of that connection has been compromised. A fresh handshake with (EC)DHE is required for better guarantees. Nevertheless, the compromise of one PSK would not lead to the compromise of another, as PSKs are uniquely driven from a long-term shared secret.

An early data transmission, i.e. 0-RTT, offers the same security features as a regular 1-RTT data with two exceptions [Res18, Appendix E.5.]. First, the transmission mode does not provide full forward secrecy as the data is encrypted with a traffic key derived from a PSK. Second, the server is not able to guarantee the uniqueness of the received record, i.e. non-replayability is not guaranteed. This presents potential harm to the website and the user.

In the case of non-idempotent records, simple duplication of action may result in side effects. An attacker could also store and replay the data to reorder it with respect to other messages. Besides, cache timing behaviour could be exploited to reveal the contents of the message by replaying it to different cache nodes and measuring latency. The application using TLS should be designed to accept only idempotent 0-RTT data to counter such attacks. Even if this is the case, a large number of replays could be exploited for timing attacks, resource exhaustion and

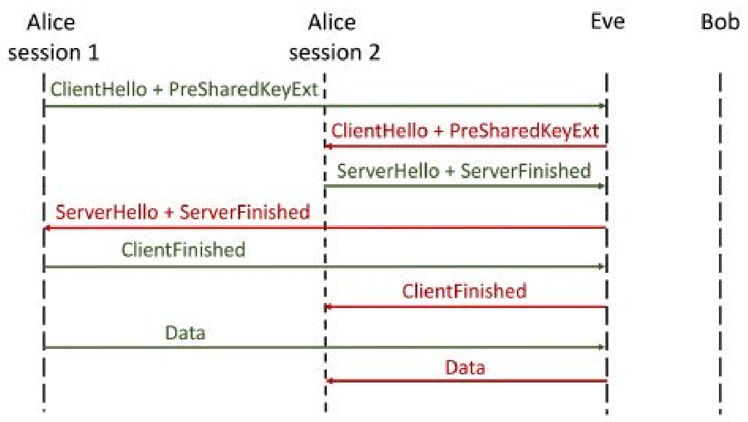

Alice session 1
Alice session 2
Eve
Bob
ClientHello + PreSharedKeyExt
ClientHello + PreSharedKeyExt
ServerHello + ServerFinished
ServerHello + ServerFinished
ClientFinished
ClientFinished
Data
Data

Figure 4.2: The SELFIE Attack. Eve tricks Alice to believe she is talking to Bob while she is actually talking with herself. [DG19]

others. The replay behaviour of a client and the inconsistent global state of a server can be used to force a server to receive multiple copies of an application message. This behaviour cannot be prevented on the TLS layer and must be handled on the application layer.

The lack of forward secrecy and non-replay guarantees for PSK and early data could be addressed by ensuring that PSKs are used only once [Res18, Section 8.1]. This would involve a database managing all outstanding valid tickets and deleting each ticket as they are used. In case of unknown tickets, the server would fall back to a full handshake. Since PSKs are always deleted upon use, connections made in only PSK key exchange mode would have forward secrecy. However, in a multi-server, distributed environment, sharing the session database between nodes poses a daunting challenge. It can also result in a lower rate of successful connections. Other recommended anti-replay mechanisms include `ClientHello` recording and `ClientHello` freshness checking. All of the above mentioned anti-replay mechanisms are associated with higher operating costs and are unlikely to be used in practice.

Apart from the weaknesses identified during the standardisation process and discussed as such in the RFC, PSK was the subject of a reflection attack called SELFIE [DG19]. The attack is applicable to both PSK key exchange modes. SELFIE presents an attack scenario in which the sender of an authentic message could be the receiver itself. The attack requires a node to open two independent connections in a network, where each node acts as a TLS server and client. As shown in Figure 4.2, an active eavesdropper can trap communications and reflect messages from a node to itself. Notably, the attacked node does not explicitly verify the identity of the node it is communicating with. Consequently, the node cannot exclude the possibility that it receives an echo of its own messages.

The attack exploits implicit assumptions made in the design and security proofs [DG19]. Namely, the protocol assumes that when using PSK, a node receiving a message knows that the message was also sent by a node that owns that PSK. Furthermore, it was wrongly assumed that PSKs could not be shared by more than two nodes, i.e. a client and a server. However, this did not take into account a scenario in which a PSK only belongs to two nodes, but the nodes can run as both client and server. In this case, a PSK can be shared by two servers and two clients in total. Besides, the fact that the protocol does not require an explicit server and client authentication in each message is used to break mutual authentication.

4.3 Interoperability

In this section, we describe how the long history of TLS and its widespread use have impacted the design of the new version. Furthermore, we attempt to answer the question to what extent the transformation of TLS has been limited by interoperability factors. With interoperability, we consider features that are not among the primary objectives of the protocol and are only present to make the protocol functional. These include backward and forward compatibility, Internet infrastructure considerations, and issues with vendors.

4.3.1 Starting from scratch

One possible solution to the interoperability problems of TLS is to start from scratch with a new protocol. A completely different protocol, rather than an improvement to the existing one, means trivially no legacy issues. Defining an entirely new set of messages and a new state machine using new mechanisms and design tools could provide a significant advantage. The much higher degree of freedom in design means that the objectives of this protocol could be achieved with less complexity. Consequently, the new protocol could ease the implementation and verification of security properties. This also could allow developers to pursue a higher (security) goal than what an improvement allows. QUIC's approach to providing the services of HTTP, TLS and TCP in a single protocol illustrates this (section 4.1.1).

Imagining this possibility puts the design limitations of TLS 1.3 into perspective and partly explains the subsequent complexity of the protocol. The immense challenge of achieving an almost entirely different set of goals while maintaining the more than two decades old TLS framework can only be understated. On the other hand, the benefits of a completely new protocol regarding security could be minimal. Problems in redesign caused by limitations in verification mechanisms or other shortcomings could also arise in the case of a brand new protocol. Also, implementation errors are not necessarily avoided. The cryptographic primitives used would mostly remain the same since TLS already uses the most secure algorithms known. Besides, creating new libraries if necessary, could be more error-prone than reusing the existing and reliable ones.

Assuming that a supposedly better protocol is developed, a significant challenge would be to achieve widespread deployment. Introducing only a new version of an existing protocol has the advantage of having several vendors who can facilitate deployment on already running servers. Only organisations such as Google and Facebook, which control both sides of the connection, could achieve a significant spread of an entirely new protocol. QUIC's track to representing a significant portion of Internet connections by taking advantage of Chrome's market share will be a pattern to monitor.

4.3.2 The theory of TLS version iteration

A key question in redesigning TLS has been how much change the protocol allows without significantly affecting interoperability. This question becomes complex given the many versions in use and the stagnant server-side implementations. As will be described later, the situation is further exacerbated by flawed middleboxes.

Ideally, TLS peers would be able to negotiate their preferred common parameters, including the protocol version, without any issue [Res18, Section 9.3]. Since the parameters to be introduced are not restricted, the protocol would be compatibly extensible, and version iteration would also be straight forward. Older protocol versions and features would be used alongside newer ones without being explicitly deprecated. The newer and better parameters would naturally overtake based on peers' preference. Support for older parameters would not be a weakness, as the protocol guarantees downgrade protection.

Version negotiation between endpoints that potentially support different versions is a critical aspect of TLS interoperability. With this in mind, `ClientHello` messages in all versions since SSLv3 have been designed to be compatible with each other [Res18, Section 9.3]. This compatibility should allow servers to handle clients trying to use newer versions as long as a commonly supported version can be found. Notably, only the `ClientHello` message is sent before a version is negotiated. Any message after that is sent according to the negotiated version. Meaning, any message beyond the `ClientHello` is allowed to change arbitrarily. In particular, the format and values contained in messages like `ServerHello` and encrypted application data should not be restricted due to interoperability.

Aside from keeping the `ClientHello` compatible, TLS follows two main principles to facilitate seamless transformation [Res18, Section 9.3]. First, a client should support all the parameters it advertises in its `ClientHello`, including supported versions, cipher suites, and extensions. Second, a server receiving a `ClientHello` should correctly ignore any unrecognized parameters. A client should also ignore any unrecognized extensions that are received as part of server messages. This way, both the client and the server could ignore any proposed or requested features they do not support yet without further impact. In theory, this should allow new features to be introduced into existing versions and more importantly, new versions to be unrestricted by the existing framework.

Unexpectedly, middleboxes have proven to be a significant challenge in redesigning TLS than the actual endpoints [Res18, Appendix D.4.]. A middlebox is any intermediary box performing functions other than the standard functions of an IP router on the data path between the client and the server. This includes load balancers, firewalls and proxies. As we will see later, most interoperability considerations were caused by such devices. In principle, middleboxes should behave like any other client or server. Moreover, a middlebox that forwards a `ClientHello` that it does not understand should not process messages beyond the `ClientHello` and should forward all subsequent traffic unchanged. Otherwise, it cannot work with newer endpoints. Middleboxes that do not follow this have experienced problems when the endpoints negotiate TLS 1.3.

4.3.3 The reality of redesigning TLS

The design of TLS 1.3 was severely limited by the non-compliance of the protocol invariants mentioned above, especially by middleboxes. Several workarounds were required to make the protocol functional. These have led to unnecessary complexity, which could indirectly affect the security of the protocol. Breaking connections to or through non-compliant endpoints and middleboxes has not been an option, as they represent a substantial part of Internet connections.

In contrast to the envisioned interoperability between versions, introducing new versions of TLS has been a challenge. In the past, flawed TLS 1.0 server-side implementations have rejected TLS 1.1 `ClientHello` [Ben]. The same has happened with TLS 1.2 `ClientHello` with flawed TLS 1.1 servers. The inability to negotiate a commonly supported version by computing the minimum of the server's and the client's maximum supported value had critical consequences. This issue of version intolerance has been addressed with version fallback, which means that the client would make a repeated attempt by disabling the rejected version or feature. Multiple fallbacks might be necessary until the connection is successfully established. Version fallback trivially bypasses the provided downgrade protection and allows attacks like POODLE.

The problem of version intolerance was once again repeated with the introduction of TLS 1.3. According to Google and SSL pulse, 1.6% to 3.2% of top sites rejected a `ClientHello` with TLS 1.3 as its maximum supported version [Ben] [Res18, Appendix D]. Since this was a significant part of the Internet and version fallback has its disadvantages, another workaround was deemed necessary. As described in chapter 3, the maximum version field in TLS 1.3 is fixed to TLS 1.2, and a new supported version extension is used for the actual version negotiation. Besides avoiding version fallback, this workaround has the advantage that individual versions can be advertised instead of a range of versions. The new approach becomes useful as clients gradually stop supporting earlier versions, and also facilitates experimentation before deployment.

After version intolerance was addressed, middleboxes presented another challenge. Field measurements showed that a significant percentage of middleboxes misbehave when peers negotiate TLS 1.3 [Res18, Appendix D]. Another design change was necessary to increase the chance of a connection via these middleboxes. The TLS 1.3 handshake was modified to resemble a TLS 1.2 session resumption starting with draft 22, also known as compatibility mode [Ben]. In particular, several dummy fields and messages to be ignored were added. These include a non-empty "legacy_session_id" field in `ClientHello`, which is echoed by the server, and a `ChangeCipherSpec` message. Similarly, the `HelloRetryRequest` message has been modified to look like a `ServerHello` to avoid upsetting the middleboxes. The compatibility mode has made the protocol challenging to understand and possibly difficult to implement correctly. However, this has not affected the actual flow and security objectives of the protocol.

The design considerations in TLS 1.3 highlight the challenge of introducing new features while keeping the old ones running, and demonstrate that the promised extensibility of the protocol is far from reality. As described above, this is mainly due to servers and middleboxes running flawed implementations. Since the problem is prevalent, effectively updating buggy servers or middleboxes is hard, if not impossible. Moreover, some of the errors may not be noticed until a specific change to the protocol is attempted. Widespread flawed implementation could also be the case with new TLS 1.3 implementations.

As the protocol's freedom to introduce new messages or change the old ones becomes more and more restricted, functionality is shifting to extensions. As a result, the guarantee that servers (and middleboxes) will correctly ignore unrecognised extensions becomes increasingly essential. Otherwise, it would be practically impossible to further enhance the protocol. One measure taken to preserve the meagre extensibility of TLS is Google's GREASE [Ben20]. By generating random extensions and including them in `ClientHello` messages from the chrome browser, it attempts to detect errors in implementations and preserve extensibility.

Chapter 5

Evaluating the trade-off in the design of TLS 1.3

Based on the many aspects of TLS 1.3 discussed so far, in this chapter, we evaluate the trade-offs made between the protocol's different sets of objectives. We name concrete compromises, comment on their necessity and significance.

5.1 Efficiency vs Security

TLS 1.3 has achieved significant efficiency goals. This with regard to reduction in both latency and computation. The protocol has also mostly addressed the security vulnerabilities of previous versions. Many improvements were made without any compromise between the two sets of objectives, such as the reduction of the regular handshake latency from 2-RTT to 1-RTT and the removal of unsafe cryptographic components. On the other hand, some efficiency goals were directly linked to critical security compromises. Namely, the further reduction of the latency to 0-RTT (early data) and avoiding public key computation using PSK (only-PSK key exchange mode). These two efficiency goals were the cause of many design considerations and have led to a conscious decision to forego certain security objectives, namely, non-replay guarantees and forward secrecy.

Design decisions to achieve higher security have been rather aggressive, as described in section 3.1. This mainly in order not to carry the shortcomings of previous versions. Accordingly, the changes in the encryption mechanisms, key exchange methods and signature algorithms have been radical. In contrast, this approach is lacking in dealing with weaknesses introduced due to efficiency objectives. This reluctance raises the question of whether efficiency has taken over security as the protocol's top priority. The design of TLS 1.3 was arguably the first time that efficiency played a central role in the design of a TLS version. Nevertheless, the design decisions suggest, the protocol's transformation has been somewhat hijacked so that the protocol cannot be as secure as it can be. For this, it suffices to imagine TLS 1.3 without PSK and early data. In such a case, the protocol would achieve all of its security goals, with all AEAD cipher suites and only (EC)DHE key exchange mode. The efficiency improvements would have also been reasonable with the 1-RTT handshake. All known weaknesses of TLS 1.3 stem from further pursuit of efficiency.

As described in Section 4.1, the urge for more performance was strong, and the new version has responded accordingly. The design decisions to adapt to modern applications and to respond to the challenge posed by other protocols are justified. On the other hand, the subsequent weaknesses arising from the security compromises in the protocol are significant, as illustrated with attack potentials disclosed during standardization as well as by the attacks made public afterwards. Nevertheless, it is crucial to realize that the protocol only offers options for endpoints to choose from. The compromises in design only mean that unsafe options are also included, but the endpoints are not obliged to carry the security compromises. Meaning, the actual trade-off between efficiency and security is pushed down from the protocol to the client and server. The endpoints have the option to have both the highest possible security as well as the best performance with security compromises. Endpoints could disable PSK key exchange modes and early data in implementations to take full advantage of the protocol's enhanced security. To further adjust the trade-off, PSK key exchange modes could be allowed, but not early data.

We believe that the design of TLS 1.3 has struck the right balance between its efficiency and its security objectives with the currently available mechanisms. Future versions, in combination with technical advancements, could minimize the trade-offs and their impact, so that more of the protocol's objectives could be achieved. For instance, reducing the computational cost of the (EC)DHE key exchange could facilitate PSK with (EC)DHE key exchange mode becoming the norm and only-PSK key exchange mode being scrapped. This modification would notably lead to forward secrecy guarantees for all key exchange modes. Besides, future versions of higher-level protocols could guarantee that the data to be sent in 0-RTT is always idempotent, thus reducing the risks of earlier data.

5.2 Security vs Interoperability

The security enhancements in TLS 1.3 did not involve significant compromises due to interoperability, partly due to the protocol's approach to relying mainly on removing unsafe mechanisms and retaining the secure once to achieve its security objectives. This is reflected in the approach to reduce the provided cipher suites to AEAD only and the public key-based key exchange methods to (EC)DHE only. The additional introduction of security features was not significantly restricted by interoperability, as was shown with new curves and new signature algorithms. The protocol was also able to improve privacy by introducing the `EncryptedExtensions` message and improve downgrade protection using the "random" field in the `ServerHello` message. In this regard, the protocol has achieved the highest level of security it is intended to achieve.

On the other hand, flawed endpoint and middlebox implementations have led to significant design restrictions. However, thanks to some ingenious solutions, the restrictions did not lead to direct security compromises. In particular, the adoption of the "supported_versions" extension against version intolerance instead of version fallback as in previous versions was essential. Notably, downgrade protection is a central aspect of the protocol's security objectives which otherwise would have been bypassed. Similarly, emulating a TLS 1.2 session resumption against middlebox problems has kept the protocol's flow and security properties intact without affecting interoperability. However, the additional complexity resulting from the required workarounds is a cause for concern. The complexity may have made it more challenging to carry out security analysis, resulting in weaknesses possibly being overlooked. Besides, as described in section 4.3, the correct implementation of previous versions, which are more straightforward compared to TLS 1.3, has already been a challenge for many vendors. The increased complexity of the new version has frightening implications, especially for future protocol modifications. In this regard, an analysis of popular TLS 1.3 implementations in a similar fashion to [BBD+17] could help further concretize the trade-offs between security and interoperability.

The measures adopted to address the trade-off between security and interoperability were necessary and very effective. Unlike the compromise between efficiency and security, this was not between two explicitly laid out sets of objectives. Instead, the challenges were discovered while attempting deployment, consequently making the measures taken ad-hoc. Besides, the existence of this particular trade-off illustrates the state of the TLS ecosystem. In particular, the actors involved in shaping the ecosystem are very diverse, some more competent than the others. As the IETF is only one of the many actors involved in shaping the protocol, this particular trade-off shows that design decisions and published RFCs have only limited influence in the wild.

5.3 Efficiency vs Interoperability

The efficiency objectives of TLS 1.3 have not been significantly restricted by interoperability. However, the limitation to introduce new messages as with the `HelloRetryRequest` being a special `ServerHello`, and the addition of dummy fields and messages due to the compatibility mode has resulted in higher bandwidth utilization. However, since reducing latency and computational costs were the main efficiency goals, this compromise is rather insignificant. On the other hand, the design constraints due to the protocol's decades-old framework may have led to a limitation of the protocol's design goals. As a result, the protocol is still lagging behind its competitors in terms of performance.

Chapter 6

Conclusion

TLS 1.3 has achieved most of the objectives it set out. As a result, the protocol is now more secure and faster than ever. Achieving these goals has been a very tricky challenge. Notably, critical design decisions had to be made when the protocol's objectives clashed, and ingenious workarounds were required. In this thesis, we contextualised the protocol's objectives, identified trade-offs between them and evaluated the measures adopted in addressing such trade-offs.

The new version of TLS has features with unsatisfactory security properties, in contradiction to the main design goal of eliminating anything unsafe in the protocol. The design decisions made in this regard illustrate the ever-increasing role of efficiency. The introduction of early data and the prominent role of PSK mark a turning point in the protocol's timeline. In particular, efficiency has become just as crucial, if not more crucial, than security.

We have determined that there is no significant trade-off between efficiency and interoperability. Thanks to the many workarounds adopted, the impact of the trade-off between security and interoperability was limited. On the other hand, the trade-off between security and efficiency played a crucial role and was at the heart of many design decisions. As a result, the protocol offers options of both highly secure as well as fast, but with security compromises. These options, in turn, push the actual compromise between security and efficiency down to the endpoints.

The design of the TLS 1.3 has illustrated the continually deteriorating state of the ecosystem. In particular, the technical incompetence of implementers is having a tangible impact on the transformation of the protocol. Notably, this issue was not new and was also present in case of prior versions. These interoperability problems have significantly increased the complexity of the new version. Which, in turn, may increase the extent of interoperability issues in the future. The restrictions could make another radical structural departure in the transformation of TLS nearly impossible without significantly affecting the interoperability of the protocol.

Bibliography

[ABP+13] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of RC4 in TLS. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 305–320, 2013.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540, 2013.

[BBD+17] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: taming the composite state machines of TLS. *Commun. ACM*, 60(2):99–107, 2017.

[BDK16] Alexandros-Apostolos A. Boulogeorgos, Panagiotis D. Diamantoulakis, and George K. Karagiannidis. Low power wide area networks (lpwans) for internet of things (iot) applications: Research challenges and future trends. *CoRR*, abs/1611.07449, 2016.

[Ben] David Benjamin. Tls ecosystem woes | david benjamin (google) | rwc 2018, year = 2018, note = Last accessed March 20 2020, url = https://www.youtube.com/watch?v=_mE_JmwFi1Y.

[Ben20] David Benjamin. Applying generate random extensions and sustain extensibility (GREASE) to TLS extensibility. *RFC*, 8701:1–12, 2020.

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and openvpn. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 456–467. ACM, 2016.

[BTPL15] Richard L. Barnes, Martin Thomson, Alfredo Pironti, and Adam Langley. Deprecating secure sockets layer version 3.0. *RFC*, 7568:1–7, 2015.

[CHH$^+$17] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1773–1788. ACM, 2017.

[CJJ$^+$19] Shan Chen, Samuel Jero, Matthew Jagielski, Alexandra Boldyreva, and Cristina Nita-Rotaru. Secure communication channel establishment: TLS 1.3 (over TCP fast open) vs. QUIC. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 404–426. Springer, 2019.

[CMTH17] Sarah Cook, Bertrand Mathieu, Patrick Truong, and Isabelle Hamchaoui. QUIC: better for what and for whom? In *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*, pages 1–6. IEEE, 2017.

[DG19] Nir Drucker and Shay Gueron. Selfie: reflections on TLS 1.3 with PSK. *IACR Cryptology ePrint Archive*, 2019:347, 2019.

[DR08] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. *RFC*, 5246:1–104, 2008.

[HARV19] Ralph Holz, Johanna Amann, Abbas Razaghpanah, and Narseo Vallina-Rodriguez. The era of TLS 1.3: Measuring deployment and use with active and passive methods. *CoRR*, abs/1907.12762, 2019.

[KJC$^+$19] Arash Molavi Kakhki, Samuel Jero, David R. Choffnes, Cristina Nita-Rotaru, and Alan Mislove. Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols. *Commun. ACM*, 62(7):86–94, 2019.

[KRA$^+$18] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G. Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of age: A longitudinal study of TLS deployment. In *Proceedings of the Internet Measurement Conference 2018, IMC 2018, Boston, MA, USA, October 31 - November 02, 2018*, pages 415–428, 2018.

[LRW$^+$17] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan R. Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan

Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 183–196. ACM, 2017.

[MS13] Christopher Meyer and Jörg Schwenk. Lessons learned from previous SSL/TLS attacks - A brief chronology of attacks and weaknesses. *IACR Cryptology ePrint Archive*, 2013:49, 2013.

[PRRJ06] Nachiketh R. Potlapally, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. A study of the energy consumption characteristics of cryptographic algorithms and security protocols. *IEEE Trans. Mob. Comput.*, 5(2):128–143, 2006.

[PvdM16] Kenneth G. Paterson and Thyla van der Merwe. Reactive and proactive standardisation of TLS. In *Security Standardisation Research - Third International Conference, SSR 2016, Gaithersburg, MD, USA, December 5-6, 2016, Proceedings*, pages 160–186, 2016.

[Res18] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.

[SGN$^+$18] Quirin Scheitle, Oliver Gasser, Theodor Nolte, Johanna Amann, Lexi Brent, Georg Carle, Ralph Holz, Thomas C. Schmidt, and Matthias Wählisch. The rise of certificate transparency and its implications on the internet ecosystem. In *Proceedings of the Internet Measurement Conference 2018, IMC 2018, Boston, MA, USA, October 31 - November 02, 2018*, pages 343–349. ACM, 2018.

[TP11] Sean Turner and Tim Polk. Prohibiting secure sockets layer (SSL) version 2.0. *RFC*, 6176:1–4, 2011.

[Vau02] Serge Vaudenay. Security flaws induced by CBC padding - applications to ssl, ipsec, WTLS ... In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, pages 534–546, 2002.

[VP16] Mathy Vanhoef and Frank Piessens. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*. USENIX Association, 2016.

[WRWH19] Konrad Wolsing, Jan Rüth, Klaus Wehrle, and Oliver Hohlfeld. A performance perspective on web optimized protocol stacks:

TCP+TLS+HTTP/2 vs. QUIC. In Phillipa Gill and Jana Iyengar, editors, *Proceedings of the Applied Networking Research Workshop, ANRW 2019, Montreal, Quebec, Canada, July 22, 2019*, pages 1–7. ACM, 2019.