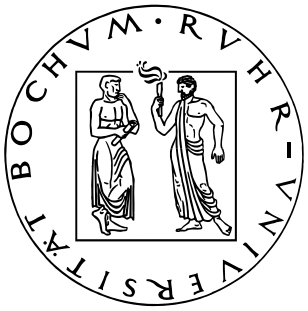
On the Security of TLS in the Real World

Robert Merget



Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs
der Fakultät für Informatik
an der Ruhr-Universität Bochum

First Supervisor:	Prof. Dr. Jörg Schwenk
Second Supervisor:	Prof. Dr. Juraj Somorovsky
External Reviewer:	Prof. Dr. Kenneth Paterson

Bochum, August 2022

Abstract

The TLS protocol is one of the most important cryptographic protocols in the world. Billions of devices use it daily to establish secure point-to-point connections that provide confidentiality, integrity, and authenticity. Since so many devices use the protocol, the protocol grew in complexity, as different requirements had to be fulfilled. This complexity makes theoretical and practical security analysis challenging, which manifests in flaws in the protocol and its implementations.

This dissertation introduces two novel attacks on the protocol, the Raccoon attack and the Alpaca attack. The Raccoon attack exploits minor timing differences in the protocol due to subtle flaws in the key derivation of TLS-DH(E) to recover the session secrets. The Alpaca attack, in contrast, exploits that TLS does not bind a TCP connection to the intended application layer protocol, which weakens the authentication of the protocol. In specific scenarios, this weakness allows an attacker to break the confidentiality of the connection.

Besides these attacks, this dissertation also introduces the newest advances in the TLS analysis framework TLS-Attacker, which are then used to build further analysis tools for either the analysis of the TLS ecosystem or vulnerability discovery. Regarding the analysis of the ecosystem, we build TLS-Scanner, a tool to evaluate the configuration of a TLS client or server and test it for common vulnerabilities and implementation flaws, and TLS-Crawler, a tool to perform large-scale evaluations with TLS-Scanner. These tools were then used to conduct studies on the TLS ecosystem regarding Raccoon, Alpaca, and CBC padding oracle vulnerabilities.

In regards to vulnerability discovery, this dissertation explores two novel approaches. The first approach is based on hybrid protocol-aware greybox fuzzing, while the other is based on protocol-state fuzzing. Both techniques could discover new flaws and vulnerabilities in widely deployed open-source TLS implementations.

Contents

1 Introduction

When the Internet was first envisioned, security was of little concern. One of the applications especially affected by this was HTTP, which did not provide any protection from Man-in-the-Middle (MitM) attackers. When the Internet got traction, the requirement for confidential, authentic, and integrity-protected communication rose, as applications like online banking were desired. To achieve this goal, Netscape developed the Secure Socket Layer (SSL) protocol, as an in-between layer between the Application layer (HTTP) and the Transport layer (TCP). The SSL protocol became one of the first major successes of modern cryptography deployed for civil society. The protocol got renamed in 1999 to Transport Layer Security (TLS) when it was adopted by the IETF. Under the IETF, the protocol evolved into one of the world's biggest and most used cryptographic protocols. The protocol is used in billions of connections worldwide every day to secure HTTP and other protocols like SMTPS [1], POP3S [2], IMAPS [2], FTPS [3], DoT [4], or DoH [5]. Additionally, TLS sparked the development of multiple related protocols like QUIC [6], DTLS [7], or OpenVPN [8]. The importance of TLS for modern Internet security is hard to overestimate. Since the protocol is so vital for the security of real-world systems, the protocol received a lot of attention from the academic community through theoretical analyses and proofs, proposals for extensions, discovered attacks on the specification and its implementations, and studies of the TLS ecosystem. TLS is such a prominent protocol, it is often used to demonstrate the effectiveness of novel analysis techniques. The hurdle for novel research is extremely high compared to other protocols, and even hard-to-exploit vulnerabilities or small implementation flaws are usually treated very seriously by the community.

The research presented in this dissertation focuses on attacks on the protocol specification, the analysis of current deployments and their properties as well as attacks on implementations of the protocol and their discovery. This work introduces two novel attacks, the Raccoon attack [9] and the Alpaca attack [10], where the Raccoon attack was directly discovered by me.

The Raccoon attack exploits a tiny timing side-channel that arises if Diffie-Hellman cipher suites are used, as the specification mandates operations that are not typically executable in constant time without extraordinary effort. In that

regard, the Raccoon attack is a rare case of a manifestation of a timing-based side channel attack in a protocol *specification*. The timing side channel leaks the most significant bits of the Diffie-Hellman shared secret, which in certain scenarios can be used to reconstruct the whole shared secret of a victim session. The vulnerability was present in the TLS specification since SSLv3 (1995) and is overall very subtle. I was able to discover the attack during my extensive work with the protocol that was conducted during the development of TLS analysis tools like TLS-Attacker. The exploited side channels are very similar to the ones in the Lucky13 attack [11] on CBC cipher suites. While the exploitation is similar to a Bleichenbacher attack [12] on RSA, the actual math used to perform the attack requires a solution to the Hidden Number Problem (HNP) [13].

The Alpaca attack in that sense is different from the Raccoon attack. The Alpaca attack exploits a subtle flaw in the authentication of TLS that can be used to mount cross-protocol attacks. These cross-protocol attacks can then be used to exfiltrate confidential data like HTTP cookies from a session. In contrast to the Raccoon attack, the potential of cross-protocol attacks on TLS was already known among some experts [14], but the issue was not well understood. With our work on the Alpaca attack, the issue was investigated, its impact evaluated, and brought to the attention of the community.

To analyze protocol implementations and deployments, it is important to have the ability to work with real-world implementations. For this purpose, different techniques can be used like the reading of source code, the development of specialized tools, or the manipulation of existing tools to fit the task at hand. In contrast to theoretical work, this kind of practical analysis is much more time-consuming as any concept has to be translated into the real world. This means that ultimately all abstractions of the conceptual model have to collapse into real messages. In a complex protocol like TLS, there are a lot of nuances that have to be accounted for by the researcher. This either limits the scope and depth of the research project or dramatically increases the required amount of time, which are of course both undesirable outcomes. To solve this issue, in 2016, Juraj Somorovsky developed a framework called TLS-Attacker [15], which allows researchers easy access to different internals of the protocol. The original version of TLS-Attacker implemented tools to demonstrate its effectiveness like a simple fuzzer, a test suite, simple tests for vulnerabilities, and proof of concept exploits. But as with previous research projects, the original version of TLS-Attacker was limited in scope as the complexity of the protocol was not tameable in all its facets in a single research project.

The work presented in this dissertation started shortly after TLS-Attacker was first released. In my role as a Ph.D. Student, I heavily extended the project and took leadership in its development. Since then, the framework grew from a single research project to a full-scale TLS analysis framework. Nowadays, TLS-Attacker supports all commonly used TLS and DTLS features and is mostly lacking niche standards that are rarely used in the real world. In that sense, TLS-Attacker often supports more TLS features than the libraries it is testing as TLS-Attacker also supports old, and knowingly broken standards. This feature-rich and versatile framework allowed me to build more advanced and complex research tools. These more sophisticated tools are not directly a part

of TLS-Attacker anymore, but due to their complexity, are stand-alone tools and merely use TLS-Attacker as a library to perform their TLS connections. The abstraction that is created by this separation allows the tools to work with TLS implementations on a much higher level than previous approaches, which in return results in higher precision tools that can uncover more aspects of the protocol implementation.

Admittedly, TLS-Attacker is working against current research trends. In other areas like fuzzing or software testing, most research focuses on techniques that are as general as possible and work on many different systems, such that the work that went into the development of the tool can be recycled as often as possible. For example, AFLnet [16] or Nyx-Net [17] are fuzzers for network protocols, and since TLS is a network protocol, these tools can also be applied to TLS implementations. The problem with such generalized approaches is that they can usually only find issues that are present in general protocol implementations and cannot find protocol-specific flaws, as the semantics of the tested protocol are unknown to the tool. For example, AFLnet can find buffer overflows in the message processing (since buffer overflows are general software flaws), but cannot find logical problems as the semantics of TLS are unknown to AFLnet. There are some noteworthy exceptions like differential fuzzing, which was successfully used by Guido Vranken in his tool 'cryptofuzz' [18]. In differential fuzzing the output of multiple different implementations is compared to each other. Ideally, all implementations should always output the same results given identical inputs and any deviation is an indicator of a logical bug. The technique works well if the desired behavior of the target is strictly defined and any deviation from the quorum can be considered a bug. In the case of TLS as a whole, this is not possible as libraries implement different subsets of the specification. A different response to inputs is therefore expected when different features are implemented. But even in identical subsets, there are usually many different responses that are considered protocol conform that do not indicate a bug. Any differential testing technique for TLS is therefore susceptible to false positives.

Since state-of-the-art techniques are limited in that regard, the creation of specialized tools seems inevitable if the highest security levels are desired. These tools are not limited by generality and can use all the context information available to find flaws as reliably as possible. TLS-Attacker is such a tool and required a specialized TLS implementation that re-implemented most of the TLS standards. This is a tremendous effort for a project that aims to analyze only a single protocol, but the importance of TLS is hard to overestimate, which justifies high-effort approaches over generalized techniques.

During my research, I was constantly searching for fitting applications of TLS-Attacker that would improve upon the state-of-the-art in dynamic protocol analysis. This search resulted in multiple very sophisticated tools and techniques that can be used to analyze TLS implementation automatically. The first two tools of this kind that are introduced in this work are TLS-Scanner and TLS-Crawler. TLS-Scanner is, as its name implies, a security scanner for TLS that extracts information about supported features, known bugs, and existing vulnerabilities from a TLS server (or client) implementation. In contrast to other scanners, TLS-Scanner can use TLS-Attacker to perform the connections, al-

lowing for rapid development without compromising in scope. Especially in the analysis of side-channel vulnerabilities, the flexibility and abstraction of TLS-Attacker prove valuable. TLS-Crawler, on the other hand, uses TLS-Scanner to extract the same information for whole groups of servers. Together, these tools allow a researcher to perform highly detailed studies about the state of the TLS ecosystem, which is interesting for researchers, developers, and standardization organizations like the Internet Engineering Task Force (IETF). In this work, three studies that I conducted with TLS-Crawler are presented. The first evaluates the security impact of the Raccoon attack, the second does the same for the Alpaca attack, and the third performs a security analysis of CBC padding oracle vulnerabilities.

Other techniques presented in this work focus on the automatic discovery of new and previously unknown vulnerabilities. The first approach in this direction is based on protocol-aware greybox fuzzing. It uses AFL-style instrumentation and the protocol awareness of TLS-Attacker to create inputs that reach deep into the implementation, even if cryptographic computations are still in place. The second approach is based on protocol state machine fuzzing and uses state-machine learning algorithms to blackbox extract a model of the implemented state-machine from an implementation. An analyst can then analyze this model to find flaws in the implementation.

Both approaches were evaluated with widely used implementations and could find previously unknown vulnerabilities. The research space with highly contextualized tools like TLS-Attacker is far from exhausted. The opportunity to work with a real-world cryptographic protocol with all its facets was previously hard to reach for researchers due to the enormous development costs of such projects.

Contributions. The following contributions were made by me:

1. I discovered a new flaw in the Finite Field Diffie-Hellman key exchange in TLS (Raccoon attack).

2. I contributed to a series of cross-protocol attacks on TLS (Alpaca attack) where I contributed to the generic TLS analysis of cross-protocol attacks.

3. I heavily extended the TLS analysis tool *TLS-Attacker*, including multiple architectural changes and feature additions that were used in multiple publications.

4. I developed *TLS-Scanner*, a security scanner based on TLS-Attacker that can test a given server or client for various properties.

5. I developed a new technique to reliably discover non-timing related side-channel vulnerabilities in real-world servers and performed a study on CBC padding oracle vulnerabilities in TLS.

6. I worked on *TLS-Crawler*, a crawler based on the developed TLS-Scanner to perform large-scale evaluations.

7. I performed large-scale studies on the Internet in regards to the Raccoon attack, Alpaca Attack, and CBC padding oracles in TLS.

8. I developed a protocol-aware greybox fuzzer based on TLS-Attacker that can use coverage information from AFL to guide protocol-aware input generation.

9. I performed a case study on TLS client fuzzing with the developed greybox fuzzer.

10. I developed a tool to perform blackbox extraction of state-machine models via protocol state fuzzing for (D)TLS implementations.

11. I contributed to an evaluation of the developed protocol state fuzzer on DTLS implementations.

Publications and Collaborations. This dissertation presents my research over the past five and a half years. This research has been, for the most part, already peer-reviewed and has been published at prestigious conferences. Concretely, this includes the following publications:

1. Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities, *Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk, and Yuval Shavitt*, **USENIX Security'19** [19].

2. Analysis of DTLS Implementations using Protocol State Fuzzing, *Paul Fiterau-Broştean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky*, **USENIX Security'20** [20].

3. Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E), *Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, and Jörg Schwenk*, **USENIX Security'21** [9].

4. ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication, *Marcus Brinkmann, Christian Dresen, Robert Merget, Damian Poddebniak, Jens Müller, Juraj Somorovsky, Jörg Schwenk, and Sebastian Schinzel*, **USENIX Security'21** [10].

5. Protocol Aware Greybox Fuzzing of TLS Implementations, *Robert Merget, Emre Güler, Phillip Görz, Juraj Somorovsky, Jörg Schwenk, and Thorsten Holz*, **not yet peer reviewed**.

Additionally, I also contributed to the following publications, which are not presented in this dissertation:

1. T0RTT: Non-Interactive Immediate Forward-Secret Single-Pass Circuit Construction, *Sebastian Lauer, Kai Gellert, Robert Merget, Tobias Handirk, and Jörg Schwenk*, **PETS 2020** [21].

2. TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries, *Marcel Maehren, Philipp Nieting, Sven Hebrok, Robert Merget, Juraj Somorovsky, and Jörg Schwenk*, **USENIX Security'22** [22].

3. Exploring the Unknown DTLS Universe: Analysis of the DTLS Server Ecosystem on the Internet, *Nurullah Erinola, Marcel Maehren, Robert Merget, Juraj Somorovsky, and Jörg Schwenk*, **to appear at USENIX Security'23**.

All of these papers were written in collaboration with my co-authors. While I contributed to many smaller aspects in each of these publications, my main contributions to each publication that appear in this dissertation are listed below.

1. **Padding Oracles.** For this publication [19], I implemented the actual padding oracle scan. The TLS-Crawler used for the evaluation was initially created by Janis Fliegenschmidt during his Bachelor thesis which I supervised and later adapted. The concrete evaluation and the analysis of the results were done by me. Nimrod Aviram, Juraj Somorovsky, and Jörg Schwenk assisted me with their rich experience in the design of the study and during the writing process. Nimrod Aviram was further supervised by Yuval Shavitt. Our research was done in parallel by Craig Young. We noticed this near the end of the publication and decided to merge the results. Craig Young's core contribution lies in the attribution and disclosure of the vulnerabilities.

2. **State Machine Fuzzing.** For this publication [20], I created the first version of a tool called *TLS-State-Vulnerability Finder*, with assistance from Joeri de Ruiter. The tool was then adapted to DTLS by Paul Fiterau-Broştean in collaboration with his colleagues Konstantinos Sagonas and Bengt Jonsson. The DTLS adaption was based on an unfinished rewrite of the DTLS protocol for TLS-Attacker by me which Paul completed. The experiments for the evaluation were done by Paul Fiterau-Broştean, while I assisted and advised him with the analysis of the extracted state machine models, especially in regards to security vulnerabilities. The project was further advised by Juraj Somorovsky.

3. **Raccoon Attack.** For the Raccoon attack [9], I discovered the initial flaw within TLS. Together with Juraj Somorovsky and Nimrod Aviram we created the first sketch for the attack. The math and the experiments around the Hidden Number Problem were done by Marcus Brinkmann, who was assisted by Johannes Mittmann after our disclosure to the BSI. The real-world timing evaluation of the hash and PRF functions, as well as the statistical computations, were done by Nimrod Aviram, while I created the data points for the Raccoon attack in a real-world setup. I further worked on the impact analysis of the vulnerability on TLS in general, especially the computations on critical block borders and the evaluation of Raccoon in the real world. The project was also guided by Juraj Somorovsky and Jörg Schwenk, especially in the writing process.

4. **Alpaca Attack.** The Alpaca attack [10] is not completely presented in this dissertation, but only the chapters that I have personally worked on. This included the generic description of the vulnerability which was created in collaboration with Marcus Brinkmann. Additionally, also in collaboration with Marcus Brinkmann, I performed parts of the Internet-wide scans for the impact evaluation of the vulnerability.

5. **TLS Fuzzing.** For this publication, I wrote the TLS fuzzer and performed the evaluation of TLS clients. The server evaluation was done by Emre Güler. The communication synchronization mechanism was developed by Philipp Görz, with the Java side of the mechanism created by me. The project was further supervised by Juraj Somorovsky, Thorsten Holz and Jörg Schwenk.

Regarding my contributions to open-source tools like TLS-Attacker, it is impossible to clearly distinguish my contributions from the contributions of my collaborators. A lot of the work on TLS-Attacker and related projects was done during Bachelor and Master theses that I supervised. I, therefore, refer to the respective Github repositories for a detailed view of the scope of my personal contributions. For the TLS-Scanner project, I developed the overall scanning architecture and most of the `Probes` in the tool. An overview of the main authorship of each `Probe` is presented in Table 8.1, but as always in bigger open-source software engineering projects, a lot of other people also contributed to the project and I again refer to the Github repository for a detailed overview.

1.1. Overview

This dissertation starts with the establishment of necessary background information. Chapter 2 introduces the necessary information from the literature to prepare the reader for the rest of the dissertation and additionally serves to establish a common language. Chapter 3 introduces TLS and gives an overview of the protocol in general. After that, Chapter 4 introduces the most important previously known attacks on TLS. After these background establishing chapters, the core of the dissertation starts with the presentation of two new attacks that I have developed or contributed to. The first attack is the Raccoon attack and is presented Chapter 5. The Raccoon attack exploits a novel timing side-channel attack that directly results from the protocol specification. In Chapter 6, the Alpaca attack is introduced, which introduces TLS cross-protocol attacks. After these chapters, TLS-Attacker is introduced in Chapter 7. TLS-Attacker is a modular framework for the practical analysis of TLS implementations which was heavily extended by me during my dissertation. Chapter 8 then introduces TLS-Scanner and TLS-Crawler, tools I developed during my research that can determine various properties of a TLS implementation. With TLS-Crawler it is then possible to perform large-scale scans. The chapter additionally presents techniques to find non-timing-related side-channel vulnerabilities in TLS implementations in the wild. The following Chapter 9, presents a series of studies of the TLS ecosystem that have been conducted with the tools that were pre-

sented in Chapter 8. After that, Chapter 10 introduces Evolutionary TLS Fuzzer (ETF), an evolutionary-based greybox fuzzer for TLS and presents an evaluation with TLS client and server implementations. Chapter 11 introduces a state machine analysis tool based on TLS-Attacker for DTLS with an extensive evaluation of the presented technique. Chapter 12 introduces the related work to this thesis, while Chapter 13 introduces directions for future research related to the presented topics. This includes future directions for further analysis regarding Raccoon and Alpaca, the TLS-Attacker ecosystem, and for general dynamic protocol analysis techniques. Finally, Chapter 14 concludes the dissertation.

2 Background

Following the explanations in upcoming chapters requires a general understanding of modern cryptography and network protocols. To avoid miscommunication, the most important fundamentals are briefly informally introduced.

2.1. HTTP

The *Hypertext Transfer Protocol (HTTP)* [23] is a line-based application layer protocol for the World Wide Web, which is typically accessed with a browser. It consists of two main components: the header and the body. The header contains metadata about the request or response, including information about the client, server, and the content being transmitted. The body contains the actual content being transmitted, such as HTML, images, or JSON data. HTTP is often used together with TLS (HTTPS) to transport the data over a secure channel. The payload of HTTP can be sensitive by itself, but besides the actual payload, HTTP traffic can also contain cookies. Cookies are used to make the otherwise stateless HTTP protocol stateful. This allows for server to implement authentication mechanism in HTTP. If a cookie gets stolen by an attacker, it oftentimes allow the attacker to impersonate a user and are therefore an attractive target for attacks.

2.2. Security Goals

When discussing IT-Security, it is important to define what security means for a given system or protocol, as security requires different definitions in different contexts. Security goals were introduced to ease communication and formally talk about security. The most important security goals for this work are presented below.

1. **Confidentiality.** Transmitted data stays secret and is inaccessible by unauthorized parties.

2. **Authenticity.** The origin of the transmitted data is authentic, meaning

that the source of the data is obvious to all communicating parties and cannot be forged.

3. **Integrity.** The transmitted data is not modifiable after it leaves its origin, without the receiver noticing.

4. **(Perfect) Forward Secrecy.** If the attacker ever gets in possession of the long-term secret, the attacker should still be unable to decrypt transmitted data.

2.3. Attackers

As already introduced in Section 2.2, it is impossible to talk about the security of a protocol without specifying what security actually means. While security goals model the properties one wants to achieve, it is impossible to achieve these properties without also modeling an attacker. An unbounded attacker can by definition always break all security assumptions. Since such a model is not helpful, the attacker's capabilities are modeled as limited. Typically, the attacker is bounded by computational and memory resources, rendering it unable to break modern cryptographic algorithms directly. Additionally, the attacker is bounded in how the attacker can interact with a given system. Oftentimes the attacker is only modeled as having access to the external interfaces of a system, while at other times the attacker is modeled as also having additional access to some internals of the system or even fictive APIs that make the attacker even more powerful.

2.3.1. Ciphertext Indistinguishability

Ciphertext indistinguishability (IND) [24] is a property of encryption schemes. To achieve IND, the ciphertext of a cryptosystem has to be indistinguishable from randomness. This ensures that even if an attacker gets access to the ciphertext it cannot learn any information about the plaintext of it, otherwise it could distinguish it from randomness. There exist different attacker models for which ciphertext indistinguishability shall be achieved. A subset of them are informally introduced below.

IND-CPA. In a Chosen-Plaintext-Attack (CPA), the attacker is given the possibility to encrypt a polynomial amount of plaintext of his choice without direct access to the key. The attacker can use this oracle to generate the ciphertext of any plaintext it desires. To achieve the IND-CPA goal, the attacker should be unable to differentiate the ciphertexts of two self-chosen plaintexts of equal length.

IND-CCA1. IND-CCA1 refers to a non-adaptive chosen ciphertext attack. In this model, the attacker has CPA capabilities and polynomial access to an oracle that answers with the plaintext to any provided ciphertext. To achieve the IND-CCA1 property, the attacker should be unable to differentiate the ciphertext of

two self-chosen plaintexts of equal length, without further access to either the encryption or the decryption oracle.

IND-CCA2. IND-CCA2 is identical to IND-CCA1, with the difference that the attacker can also access the encryption and decryption oracle after the challenge was presented to the attacker, with the only limitation that the attacker may not request the plaintext for the challenge ciphertexts.

2.3.2. Web Attacker

The web attacker is one of the most common attacker models. The web attacker is an unprivileged attacker that is attacking a system from the distance. The web attacker can send data to the system (e.g. the protocol participants) and can receive data from the peers that is directed to them. Additionally, the web attacker can set up its own servers and act as a (malicious) service provider. Effectivly, the web attacker is a normal Internet participant.

2.3.3. Man-in-the-Middle Attacker

The Man-in-the-Middle (MitM) attacker is a more privileged attacker than the web attacker. The MitM attacker has the same capabilities as the web attacker but is additionally able to access data that is transmitted from and to other peers that are not intending to communicate with the attacker. The attacker can read or modify the transmitted data, but cannot directly break otherwise secure cryptographic primitives. Often times MitM attackers are differentiated into active and passive MitM attackers. An active MitM can read and modify transmitted data, while the passive MitM can only read the data. This distinction is often made because of the technical complexity of active attacks that do not scale so well for large-scale adversaries.

2.3.4. Man-in-the-Browser Attacker

The Man-in-the-Browser (MitB) attacker was first introduced in the context of TLS by Duong and Rizoo [25] and drastically influenced attacks on the TLS protocol. The MitB attacker can execute javascript in the browser of a victim in the context of an attacker-controlled website (that is distinct from a victim website). The Same-origin Policy (SOP) in the browser prevents the attacker from directly accessing sensitive information, but simultaneously allows the attacker to send partially controllable requests from the victim's browser to a target server. An attacker typically can become a MitB attacker by tricking the victim into visiting a website under the attacker's control. The MitB attacker is often modeled as having also capabilities of a MitM attacker(MitB+MitM). While the MitB+MitM attacker seems like a very privileged attacker, that is unlikely to appear in the real world, it is surprisingly easy to become a MitB in the context of HTTP if an attacker is already a MitM, since any HTTP request that is sent from the browser can be answered by a MitM attacker to include malicious javascript, immediately granting the MitM attacker MitB privileges.

The MitM and MitB attackers are closely related to the theoretical capabilities of the attackers from Subsection 2.3.1.

2.3.5. Co-located Attacker

The co-located attacker is a very privileged attacker. The co-located attacker can execute code on the same physical machine as the victim, but cannot access the secrets of the victim directly. This attacker model seems contrived at first glance, but since cloud applications and shared hosters are emerging more and more, it is theoretically possible that a Virtual Machine (VM) of a victim is deployed on the same physical machine as the VM of an attacker. Ristenpart et al. [26] presented techniques that allow an attacker to influence the position of the attacker in a cloud. A co-located attacker shares hardware resources with the victim, which allows the attacker to retrieve internal information about a running process via various side-channels e.g. the CPU cache. Co-location also allows an attacker to measure timing-based side channels much more precisely than any network attacker could.

2.4. Symmetric Encryption

A symmetric encryption scheme is a set of algorithms that can be used to encrypt and decrypt data. A formal definition can be found in Bellare et al. [27, p.93]. Most notably, in a symmetric encryption scheme, the same key used to encrypt data is also used to decrypt data. Modern cryptography typically differentiates between two kinds of symmetric algorithms: Block ciphers and stream ciphers.

Stream Cipher. Stream ciphers are ciphers that produce a pseudorandom key stream that is then combined with the plaintext to produce the ciphertext. The plaintext is retrieved by combining the ciphertext with the (same) keystream. As a combination function, the XOR operation is typically used. Among the most prominent stream ciphers are RC4 and ChaCha [28].

Block Cipher. A block cipher is an encryption algorithm that can only process a fixed amount (a block) of data at a given time. If less data than a multiple of the block length has to be encrypted, the input is padded to a multiple of the block length. Prominent block ciphers are AES, (3)DES, Camellia, IDEA, Seed and RC2.

2.5. Cipher Mode of Operation

Since block ciphers are always only encrypting a single block, block cipher operations have to be chained together to encrypt longer messages with so-called 'cipher modes'. Depending on how these functions are used, different properties emerge. The most important cipher modes are introduced below.

ECB. The Electronic Code Book (ECB) mode is effectively the lack of an encryption mode. In ECB mode, each block is encrypted and decrypted individually without other influences. This leaks the relation of the plaintext blocks to each other. A visualization of this effect can be seen in Figure 2.1.

(a) Plaintext (b) AES-ECB (c) AES-CBC

Figure 2.1.: Visualization of the effect of encryption in ECB mode. While each block is encrypted, the relations between the blocks are still visible. In CBC mode the effect disappears.

CBC. In Cipher Block Chaining (CBC) mode, each plaintext block is XOR'ed to the previous ciphertext block before being encrypted by the block cipher. Formally, let us denote plaintext message blocks by m_i, where $i = 0, \ldots, n$. We denote the encryption with a block cipher under key k as $Enc_k(\cdot)$. Then, we compute ciphertext blocks as $c_i = Enc_k(m_i \oplus c_{i-1})$. The above holds for all blocks except the first one, where there is no previous ciphertext block – instead, that block is XOR'ed with an initialization vector (IV) before encryption: $c_0 = Enc_k(m_0 \oplus IV)$. The decryption is performed in reverse by computing $m_i = Dec_k(c_i) \oplus c_{i-1}$

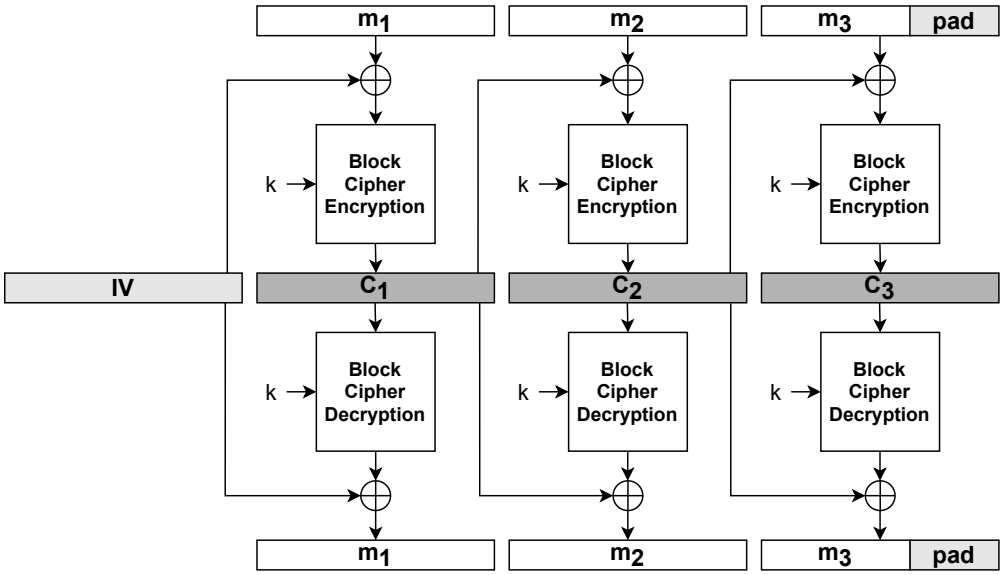


Figure 2.2.: Encryption and decryption with the CBC mode.

CM. In Counter Mode (CM), a counter i is encrypted and incremented for each block of plaintext. Then, each encrypted counter is XOR'ed with the corresponding plaintext block and output as ciphertext. The plaintext is not padded and superfluous bytes are discarded.

GCM. The Gallois Counter Mode (GCM) is an Authenticated Encryption with Associated Data (AEAD) mode [29]. In GCM, each plaintext block is first encrypted with CM. Then the associated data is then combined with the ciphertext to create a Galois Message Authentication Code (GMAC) that authenticates and provides integrity for the ciphertext. A receiver can then use the GMAC to validate the integrity and authenticity of the ciphertext. If the GMAC is valid, the receiver can proceed to decrypt the ciphertext.

CCM. The Counter with CBC-MAC (CCM) mode is another AEAD mode. It uses CM mode for the encryption and then attaches a CBC-MAC [30] to the ciphertext which uses the same encryption key as the CM.

2.6. Hash Functions

Hash functions are mappings $h : \{0,1\}^* \rightarrow \{0,1\}^N$ which are one-way, collision-free and do not allow to compute second preimages [31]. Most common cryptographic hash functions are built using a Merkle-Damgård construction [31] (see Figure 2.3). In this construction, the input is split into fixed-size blocks, and each block is mixed into a state of the computation using a compression function until all blocks have been processed. Before feeding the blocks to the compression function, the input is extended by a length field, and then padded to a multiple of the block size of the hash function. The extension and padding may necessitate creating an additional input block. In some constructions, the output is fed to a finalization function, which compresses the internal state to the final output.

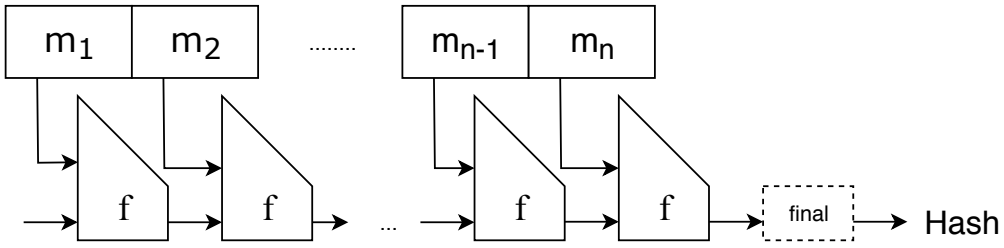


Figure 2.3.: Merkle-Damgård construction of common hash functions, such as MD5, SHA1, and SHA2.

2.7. MAC

A Message Authentication Code (MAC) is a cryptographic tag used to authenticate and integrity protect a message. The MAC is computed with a function

with a message and a symmetric key as input. Anybody possessing the key and the message can use the MAC to verify that the message has not been tampered with by somebody without the key.

2.7.1. HMAC

HMAC is a mechanism to compute message authentication codes based on hash functions. The HMAC can be instantiated with any hash function H and then inherits the parameters of H. For example, HMAC-SHA1 has an internal block size of 64 bytes and an output size of 20 bytes. HMAC uses the following construction:

$$\mathrm{HMAC}_H(K, M) = H\Big((K \oplus opad)||H\big((K \oplus ipad)||M\big)\Big)$$

Here K is a secret key, and *opad* and *ipad* are byte arrays of hash input block size B filled with bytes `0x36` and `0x5C`, respectively. The secret key K must also have a fixed length B. Therefore, before computing the HMAC, K is either padded with zeros (if $|K| < B$) or hashed with the hash function H (if $|K| > B$).

2.8. Pseudo-Random Functions

A pseudorandom function (PRF) is a family of functions that produce pseudo-random output. A key is then used to select a function from the family. The output of each function should be indistinguishable from a truly random function and it should be impossible to retrieve the key (and therefore know which function was selected). A formal definition can be found in Bellare et al. [27, p.59].

2.9. HKDF

The HMAC-based key derivation function (HKDF) [32] is a PRF specifically designed for key derivation. The construction uses an expand-then-extract technique which is designed to efficiently spread the entropy from the input key to the output.

2.10. RSA

RSA [33] is a widely used asymmetric cryptographic scheme. The RSA cryptosystem works as follows:

1. Alice chooses two large random primes (typically 1024-bit long) p and q.
2. Alice computes a modulus $N = p * q$
3. Alice computes $\varphi(N) = (p-1) * (q-1)$
4. Alice chooses a public key e such that $1 < e < \varphi(N)$ that is coprime to $\varphi(N)$. In practice e is typically chosen as $e = 2^{16} + 1$. Note that small(er) values for e can be insecure under certain conditions (e.g. $e = 3$ [34, 35]

5. Alice computes $d \equiv e^{-1} \pmod{\varphi(N)}$

After these steps, the initialization of RSA is completed. d is the private key only known to Alice, while e and N are the public key, which can (and should) be public. If Bob wants to send an encrypted message to Alice, Bob can use Alice's public key by choosing a plaintext message $0 <= m < N$. Bob then computes $m^e \pmod{N} \equiv c$ where c is the ciphertext. Alice can then compute the plaintext as $m \equiv c^d \pmod{N}$.

RSA as defined here (also called textbook RSA) has some noteworthy properties which are worth mentioning here. First off, RSA is malleable, an attacker can manipulate the ciphertext to produce a controllable manipulation of the plaintext: $(c * e^s)^d \equiv m * s \pmod{N}$. Additionally, textbook RSA allows the attacker to guess the message m. An attacker can simply encrypt its own guesses for m and see if the ciphertext is identical. In textbook RSA, the same message will always encrypt to the same ciphertext, creating a similiar effect to the ECB mode (Figure 2.1). To prevent these (and other) issues RSA should always be used with a padding scheme and never in its raw form. Common RSA padding schemes are PKCS#1 v1.5 [36] or OAEP [37].

2.11. Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange protocol [38] (also called finite field Diffie-Hellman or DH) is a key exchange protocol belonging to the group of asymmetric cryptosystems. The protocol works as follows:

1. Alice and Bob have to agree on a big prime number p and a generator g.
2. Alice and Bob each generate a random number a (Alice) and b (Bob) which they keep secret (their private keys).
3. Alice computes $g^a \pmod{p}$ while Bob computes $g^b \pmod{p}$ as their respective public keys.
4. Both then exchange their public keys
5. Both parties can then compute a shared secret as:
   $(g^a)^b \equiv (g^b)^a \equiv g^{ab} \pmod{p}$

An attacker who can only see the transmitted public keys $g^a \pmod{p}$ and $g^b \pmod{p}$ can not compute g^{ab} without computing the discrete logarithm which is considered hard.

2.12. Elliptic Curve Diffie-Hellman Key Exchange

In contrast to the finite field Diffie-Hellman key exchange, there exists also a variant of the protocol that is based on elliptic curves (called ECDH). In this variant, the generator is a point on an elliptic curve (the base point). The public keys and the shared secret, therefore, are also points on the elliptic curve. The elliptic curve Diffie-Hellman key exchange is typically faster than its finite field

counterpart. The generator and the elliptic curve on which the computations are performed are non-trivial to choose securely. Therefore, typically only a few carefully selected parameter combinations are chosen from.

2.13. Signatures

A digital signature is a cryptographic checksum that can be used to verify the integrity and authenticity of data. A digital signature scheme belongs to the family of asymmetric crypto schemes. Digital signature schemes have two keys, a public, and a private key. The private key is used to create the digital signature, while the public key is used to verify the signature.

RSA. The RSA [33] cryptosystem is naturally also a digital signature scheme. To sign data with RSA, Alice uses her private key to sign the message, as if she were 'encrypting' it with her private key by computing $m^d \equiv s \pmod{N}$. Bob can then verify the signature by 'decrypting' the signature and checking that the decrypted value equals the message m: $s^e \equiv m' \stackrel{?}{=} m \pmod{N}$. As with RSA encryption, RSA signatures also require a padding scheme.

DSA. The Digital Signature Algorithm (DSA) [39] is a signature scheme by NIST based on the discrete logarithm problem.

ECDSA. Elliptic Curve Digital Signature Algorithm (ECDSA) [40] is a DSA-inspired signature scheme based on elliptic curves.

2.14. Hidden Number Problem

Boneh and Venkatesan presented the Hidden Number Problem (HNP) [13] in 1996, originally to show that using the most significant bits (MSB) of a Diffie-Hellman secret is as secure as using the full secret. Their proof includes an algorithm that, given an oracle for the MSBs of DH shared secrets where one side of the key exchange is fixed, computes the entire secret for another such key exchange. The algorithm presented in that seminal work uses basis reduction in lattices to efficiently solve the Closest Vector Problem. While initially presented as part of a positive security result, the HNP and its solutions later were also used as components in cryptographic attacks. For example, such algorithms have been used to break DSA, ECDSA, and qDSA with biased or partially known nonces [41–48].

To solve the *Hidden Number Problem* (HNP) [13], an adversary must compute a secret integer α modulo a public prime p with bit-size n, given information about the k most significant bits (MSBs) of the n-bit representation of random multiples $\alpha \cdot t_i \bmod p$ of this secret value. From these MSBs the adversary can construct integers y_i (e.g., by setting the MSBs of y_i as the known bits, and all other bits to 0) such that for each i we have $0 \leq \alpha \cdot t_i \bmod p - y_i < p/2^\ell$ for some $\ell > 0$. Each triple (t_i, y_i, ℓ) contains ℓ bits of information on α. The number

$\ell := k - n + \log_2(p) \in [k-1, k]$ can be considered the effective number of given MSBs. This number can also be written as $\ell = k - \epsilon$, where $\epsilon = n - \log_2(p)$ represents the bias of the modulus. If ℓ is not too small and we have a moderate number of equations, the hidden number α can be recovered by solving an instance of the Closest Vector Problem (CVP) in a lattice [13, 47, 49]. If ℓ is small and a large number of equations is available, Fourier analysis is considered more promising [50, 51].

3 Transport Layer Security

Transport Layer Security (TLS) [52] is a cryptographic layer between the transport layer (i.e., TCP) and an application layer protocol and is one of the the most prominent cryptographic protocols in the world. It is used to secure HTTP connections from browsers to web servers and is therefore effectively integrated on every consumer device. The protocol was designed as a semi-transparent layer that can be used to easily upgrade legacy protocols that did not integrate cryptography from the start, like SMTP, POP3, IMAP, and FTP.

In its default configuration, TLS provides confidentiality, integrity, and authenticity. TLS was first developed under the name Secure Socket Layer (SSL) in 1994 and was renamed in 1999 to TLS when it was adapted and standardized by the Internet Engineering Task Force (IETF). TLS has many users with different requirements, which caused the standard to grow immensely over the last 25 years ago. At the time of writing, the TLS protocol is spread across over 50 TLS specific RFCs, excluding RFC's from other working groups like PKI, ASN.1, X.509 or basic cryptographic primitives. This created a complex ecosystem that was subject to various attacks and vulnerabilities in the past and is frequently updated with new versions, extensions, and standards to be more secure and robust. The latest version of the protocol is TLS 1.3 [53], while the older versions TLS 1.0, 1.1, and 1.2 [52, 54, 55] are typically still deployed alongside of it. The older versions SSLv3 and SSLv2 are considered insecure and should not be used anymore. SSLv3 and the TLS versions 1.0 to 1.2 all share a similar structure, while TLS 1.3 overhauled the protocol's design and is fundamentally different from the previous versions. The first officially released version, SSLv2, is structurally different from all other TLS versions and has severe weaknesses. This work will focus on SSLv3 up to version TLS 1.3.

3.1. Protocol Structure

The TLS protocol is structured into two layers, the record layer and the message layer, where the message layer consists of several sub-protocols (see Figure 3.1). The record layer wraps the data from the message layer into records that act like an envelope. Each record contains information about the desig-

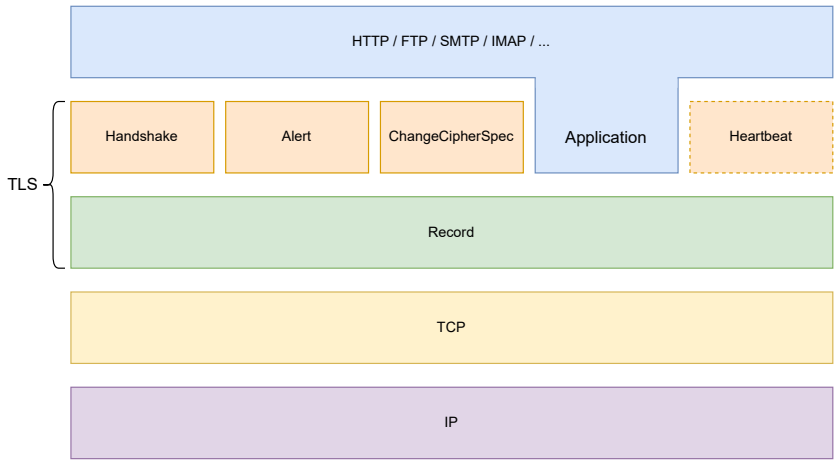


Figure 3.1.: The architecture of the TLS protocol.

nated message layer protocol, the negotiated TLS version, and the length of the transmitted payload. The message layer consists of the handshake protocol, the change cipher spec protocol, the alert protocol, and the application protocol, with additional optional protocols. The handshake protocol is used to negotiate cryptographic material, algorithms, and other TLS features securely. The change cipher spec protocol is used to communicate that the negotiated parameters of the session are now activated. The alert protocol is used to communicate errors during the protocol execution, while the application protocol is used to transmit the actual application data from the application layer. The application data protocol may only be used once the TLS handshake has been completed at least once. A TLS handshake involves different messages from the handshake and change cipher spec protocol and follows strict rules, where the concretely exchanged message can change depending on the negotiated parameters, as well as the configuration of the server and the client.

3.2. Handshake

The TLS Handshake was significantly changed from TLS 1.2 to TLS 1.3. While in the future, the most important protocol will be TLS 1.3, we will first focus on the handshake of TLS 1.2 and older versions, as some of the later introduced changes appear out of place without a deep understanding of the history of the protocol. The handshake of TLS 1.3 will be introduced in Section 3.15.

To simplify the explanation, we will first focus on the most common message flow and introduce notable exceptions later. Each TLS handshake begins with the client sending a `ClientHello` message, which is typically responded to by the server with a `ServerHello` message. With the `ClientHello`, the client communicates the highest protocol version it supports, a list of cryptographic algorithm combinations called cipher suites, a nonce (called `ClientRandom`), a list of supported compression algorithms, an optional session ID, as well as a list of extensions. The cipher suites are a concrete selection of

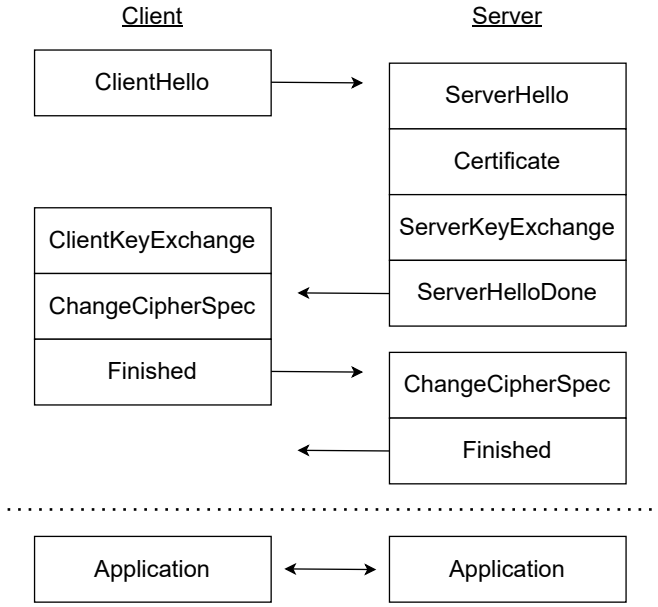


Figure 3.2.: A typical TLS 1.2 handshake. After the handshake, both peers can securely exchange application data.

algorithms for all required cryptographic tasks. For example, the cipher suite `TLS_DHE_RSA_WITH_AES_128_GCM_SHA256` uses an ephemeral DH key exchange with RSA signatures to establish a shared session key. In order to encrypt and authenticate data, it uses symmetric AES-GCM encryption with a 128-bit key, while SHA256 is used as a hash function in the key derivation. While the client primarily communicates its supported features with its `ClientHello`, the server mostly chooses a set of features that should be active for the session with its `ServerHello` from the parameters the client provided. With the `ServerHello`, the server transmits the selected cipher suite and version, a nonce (called `ServerRandom`), a session ID, as well as a compression algorithm and a list of TLS extensions to the client. The server follows this message up with a `Certificate` message, which contains an X.509 certificate chain. The leaf certificate contains the server's public key that can either be used for the key exchange directly or to validate signatures created by the server. Depending on the cipher suite, the server then sends a `ServerKeyExchange` message, which contains an ephemeral public key, as well as a signature, generated with the private key of the leaf certificate. The signature signs the parameters in the `ServerKeyExchange` message, as well as client and server randoms. The server then sends a `ServerHelloDone` message, which signals to the client that the server has finished its flight. The client then sends a `ClientKeyExchange` message containing the client's public key. Both parties now have the cryptographic material to compute a shared secret called the premaster secret (PMS). The PMS is then used to derive the master secret (MS) using a key derivation function (see Section 3.6); the MS is used to derive the individual symmetric keys. The client then sends a `ChangeCipherSpec` message, indicating to the server that the fol-

lowing messages sent from the client to the server will be encrypted. The last message sent by the client within the handshake is a `Finished` message, which contains a cryptographic checksum over the transcript of the connection. The server answers this with its `ChangeCipherSpec` message, indicating that from now on, all messages are encrypted, followed by the server's `Finished` message.

3.3. Change Cipher Spec Protocol

The `ChangeCipherSpec` protocol consists only of a single message sent during the handshake. The `ChangeCipherSpec` message only contains a single byte `0x01`. A peer sending this message indicates that all its following messages will be encrypted under the most recently negotiated keys. A peer receiving this message will start to decrypt all messages after it.

3.4. Alert Protocol

The alert protocol is a simple protocol to communicate errors during a connection. Each alert message contains two constants, a 1-byte constant indicating the severity of the error (e.g., warning or fatal), and another 1-byte constant that communicates a description of the alert message (for example `HANDSHAKE_FAILURE`). A notable special alert is the `CLOSE_NOTIFY` alert, which a peer can send to indicate the end of a connection.

3.5. Application Data Protocol

The application data protocol is used to transmit the actual application data from the upper layer. The protocol does not carry any additional meta information.

3.6. Key Derivation

To protect application data, TLS does not directly use the shared secret established via the key exchange algorithm but uses a key derivation function to compute symmetric keys from the shared secret. In TLS 1.2, the shared secret (called PMS) is put into a PRF together with an ASCII String (called label), as well as the client random and server random to produce the MS. The MS is then used to derive a *key block* by applying a PRF to the MS, another label, and the server and client random (in reverse order). The key block is a concatenation of all the symmetric key material needed for the connection. The key material includes the keys for the symmetric ciphers, the key for the MAC, and the IVs. If a cipher suite does not use specific keys, their length is considered zero. The key derivation is visualized in Figure 3.3.

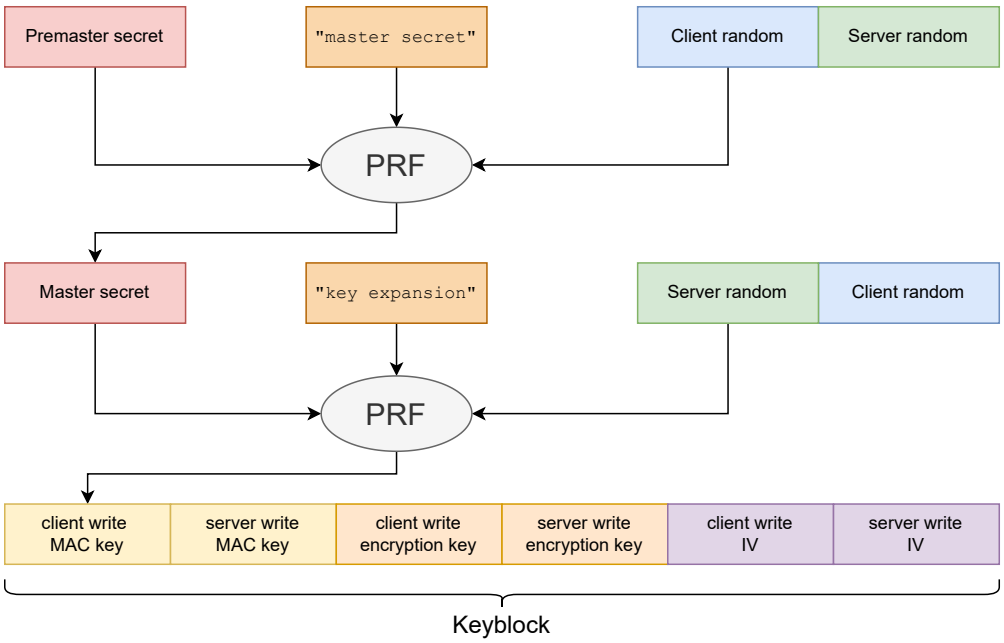


Figure 3.3.: The KDF of SSLv3 up to TLS 1.2. If a specific aspects of key block are not required for a given cipher suite, they are omitted.

3.6.1. TLS-PRF

TLS uses a custom constructed PRF, which is different for SSLv3, TLS 1.0/1.1, and TLS 1.2. The construction of the PRF in TLS 1.2 is based on HMAC. Per default, TLS 1.2 uses SHA256 as the hash function within PRF, which can be overwritten if the cipher suite specifies a stronger hash function (e.g., SHA384). The PRF in TLS uses a single hash function H, a secret K, a label, and a seed to expand cryptographic material [52]:

$$\begin{aligned}\mathrm{PRF}(K, \mathit{label}, \mathit{seed}) = &\mathrm{HMAC}_H(K, A_1 \mid\mid \mathit{label} \mid\mid \mathit{seed}) \mid\mid \\ &\mathrm{HMAC}_H(K, A_2 \mid\mid \mathit{label} \mid\mid \mathit{seed}) \mid\mid \\ &\mathrm{HMAC}_H(K, A_3 \mid\mid \mathit{label} \mid\mid \mathit{seed}) \mid\mid ...\end{aligned}$$

where $A_0 = \mathit{label} \mid\mid \mathit{seed}$ and $A_i = \mathrm{HMAC}_H(K, A_{i-1})$. Here the *label* is a distinguishing ASCII string constant defined in the TLS standard. The number of PRF iterations depends on the desired output length. For example, three iterations can be used to produce up to 96 output bytes if SHA256 is used.

In TLS 1.0 and TLS 1.1, the PRF looks similair to the PRF in TLS 1.2, with the exception that the secret is split into two halfs, and the PRF output of each half is computed individually, once with SHA1 and once with MD5. The output of those PRF executions is then XOR'ed to produce the final PRF output. This construction was chosen, because at the time of the creation of the PRF, it was unclear whether either SHA1 or MD5 would stand the test of time. The chosen design would ensure that the constructed PRF would stay secure as long as either hash function stays secure.

The SSLv3 protocol also uses a hybrid design based on SHA1 and MD5. Here the secret is first hashed with SHA1 together with a label `A` and the client and server random. This hash is then hashed again with MD5 and the secret to produce the output of the PRF. If more bytes are required, the output is concatenated with another invocation, where the label is incremented and extended in length by one (e.g. "BB", "CCC", "DDDD"...).

$$\begin{aligned}\mathrm{PRF}(K, R_C, R_S) = &\mathrm{MD5}(secret \,||\, \mathrm{SHA1}(secret \,||\, "A" \,||\, R_C \,||\, R_S) \,||\\ &\mathrm{MD5}(secret \,||\, \mathrm{SHA1}(secret \,||\, "BB" \,||\, R_C \,||\, R_S) \,||\\ &\mathrm{MD5}(secret \,||\, \mathrm{SHA1}(secret \,||\, "CCC" \,||\, R_C \,||\, R_S) \,||\, ...\end{aligned}$$

3.7. Key Exchange Algorithms

When using the TLS protocol there are multiple options for the key exchange protocol that are negotiated within the cipher suite.

3.7.1. RSA Key Exchange

If an RSA algorithm is selected as the key exchange algorithm, it is implicitly also used to authenticate the server. RSA can only be used if the server has a certificate that contains an RSA public key. If RSA is used, the server does not send a server key exchange message. The client then chooses a 48-byte long PMS, where the first two bytes of the PMS are the highest protocol version supported by the client (as advertised in the `ClientHello` message). It then applies a PKCS#1 v1.5 [36] padding to the PMS and encrypts it with the public key of the server from its certificate by computing $y = PKCS\#1.5(PMS)^e \pmod N$ and transmits this encrypted value in the `ClientKeyExchange` message to the server. The server can then decrypt the message with its private key by computing $y^d = PKCS\#1.5(PMS) \pmod N$. Both parties are now in possession of the PMS and can proceed with the key derivation as described in Section 3.6.

3.7.2. Ephemeral Diffie-Hellman Key Exchange

The ephemeral Diffie-Hellman key exchange requires the server to send a `ServerKeyExchange` message after its certificate message to transmit ephemeral public key parameters. To do this, the server first has to choose Diffie-Hellman parameters. These parameters consist of a generator g, a modulus p, a private key b, a public key $g^b \pmod p$, and a signature that signs these parameters with the private key for the presented certificate together with the client and server random. The client then chooses a private key a and computes $g^a \pmod p$ and sends this value to the server in its `ClientKeyExchange` message. Now the client can compute $(g^b)^a \pmod p$ while the server can compute $(g^a)^b \pmod p$ resulting in both parties computing the shared secret as g^{ab}. This shared secret is then used as the PMS where **leading zero bytes are removed** from it. Both parties then continue with the key derivation. DHE cipher suites offer perfect

forward secrecy. While the server is supposed to choose a fresh private key b for every incoming connection, some servers in the real world often reuse the same private key b for multiple connections to avoid recomputing $g^b \pmod p$ for every incoming client. This behavior directly conflicts with the goal of perfect forward secrecy, as the ephemeral session key is reinterpreted as a semi-long-term key.

3.7.3. Ephemeral Elliptic Curve Diffie-Hellman

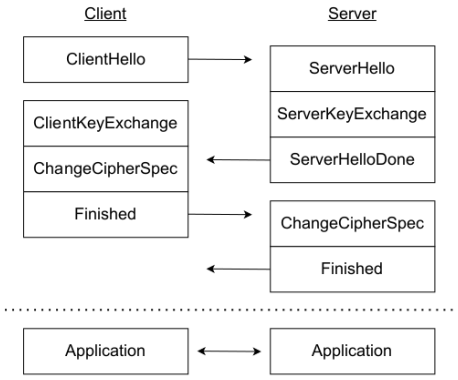
The ephemeral ECDH key exchange is similar to the finite field DH key exchange but uses elliptic curve cryptography instead. Technically, TLS supports elliptic curves with named groups (and known secure parameters) and with custom groups. While both concepts were originally specified, only the named groups were ever implemented in real-world implementations, as choosing secure elliptic curve parameters is nontrivial. In case of a named group, the server sends an identifier of the named group, a public key $g*b \pmod p$, and a signature (analog to DHE) to the client where the elliptic curve parameters are implicitly known to both parties through the exchanged identifier. The client then chooses a private key a. Now the client can compute $(g*b)*a \pmod p$, while the server can compute $(g*a)*b \pmod p$ resulting in both parties computing the shared secret as $g*a*b \pmod p$. The x-coordinate of the shared point is then used as the PMS where leading zero bytes of the x-coordinate are preserved. Both parties then continue with the key derivation. Similar to DHE, some servers in the real world reuse the same private key b for multiple connections to avoid recomputing $g*b \pmod p$ for every incoming client. As with DHE, this behavior makes the cipher suite lose perfect forward secrecy.

3.7.4. Static EC(DH) Key Exchange

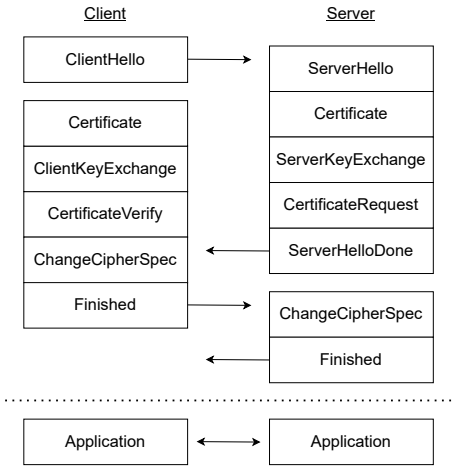
Both the ECDH key exchange and the finite field DH key exchange have non-ephemeral variants within TLS that do not offer perfect forward secrecy. In these cipher suites, the server does not transmit a new public key in a `ServerKeyExchange` message, but the server's certificate directly contains the public key. These cipher suites are not very commonly used in real deployments due to their inflexibility, as they usually require that the server possesses multiple certificates in case a client does not support the required cipher suite or group. RSA and ECDSA offer more flexibility in that regard.

3.7.5. Anonymous Key Exchange

A special group of key exchange algorithms are anonymous (anon) cipher suites. These cipher suites do not use an authentication algorithm. If an anon cipher suite is used, the server omits the certificate message(Figure 3.4a) and sends its public key in a server key exchange message. From there on, the handshake proceeds as usual. Since the server never proves its identity, anon cipher suites do not provide authentication and are vulnerable to simple MitM attacks.

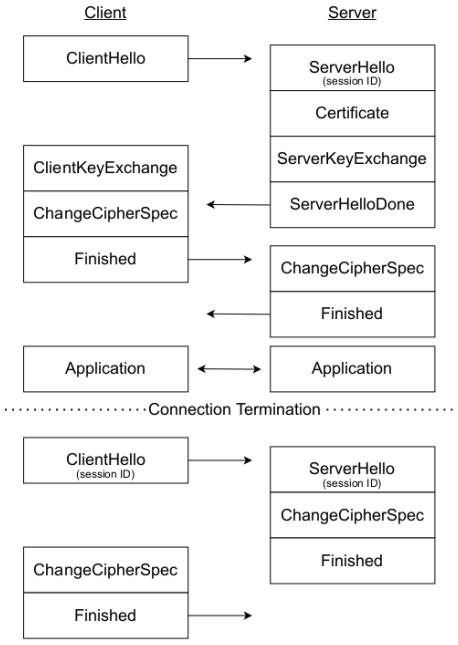


(a) A handshake with an anonymous cipher suite.

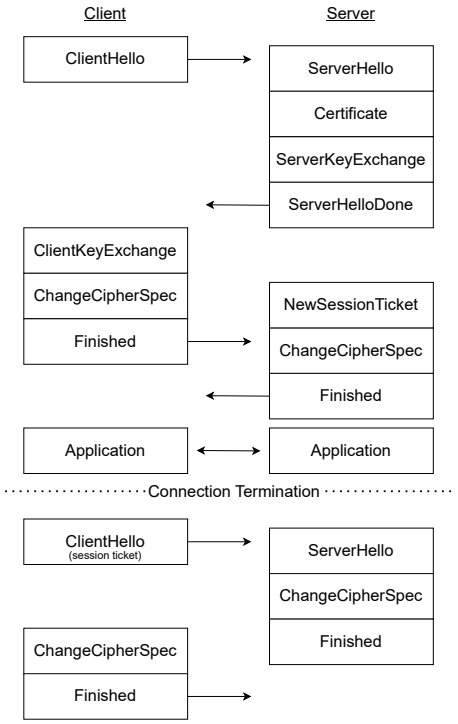


(b) A TLS 1.2 handshake with client authentication.

Figure 3.4.: TLS Handshakes without server authentication or with client authentication.



(a) A TLS session resumption with session IDs.



(b) A TLS session resumption with session tickets.

Figure 3.5.: Session resumption mechanism within TLS.

3.7.6. Other Key Exchange Algorithms

The TLS protocol also specifies other, more exotic key exchange algorithms. Among them are key exchange algorithms based on pre-shared keys (PSK), the secure remote password protocol (SRP), Dragonfly (ECCPWD), or non-NIST algorithms like the Russian GOST algorithms or the Chinese SM cipher suites. Recent efforts also experiment with post-quantum secure key exchange algorithms. Not all of these algorithms are officially specified in a full RFC. Some never made it out of the RFC-draft phase, while others were never officially documented. Still, sometimes these algorithms are implemented and deployed in the real world.

3.8. TLS Record Layer

The record layer is a protocol layer within TLS that is used to encapsulate protocol messages. In essence, the record layer wraps the protocol message with a header containing the message length, message type, and protocol version. Once `ChangeCipherSpec` messages are exchanged, subsequent TLS records will encrypt encapsulated messages. Each record has an implicit sequence number that both peers count in each direction. The sequence number influences cryptographic computations such that the order of records cannot be changed once a cipher is active. The sequence number resets whenever a `ChangeCipherSpec` message is sent in the respective direction. Any data that should be encrypted by the record layer is first fragmented into smaller bits. Each fragment can contain up to $2^{14} - 1$ bytes. Implementations do not need to use this hard limit but can also use smaller fragments. Each fragment then is optionally compressed using the negotiated compression algorithm. Modern TLS implementations typically do not support TLS compression anymore due to security concerns (see Section 4.2). After that, each record is encrypted using one of three different cipher types:

- `BLOCK`
- `STREAM`
- `AEAD`

Depending on the selected cipher type, the encryption and decryption procedures change.

BLOCK Ciphers. Despite their name, not all block ciphers are considered `BLOCK` ciphers by TLS. Due to legacy reasons, `BLOCK` ciphers refer to ciphers that use the CBC mode and use a MAC to protect the authenticity of TLS records. The TLS specification prescribes the *MAC-then-Pad-then-Encrypt* mechanism [52]. To encrypt a message, one first computes a MAC, concatenates the MAC to the plaintext, pads the message such that its length is a multiple of the block length, and then encrypts it with a block cipher in CBC mode. The

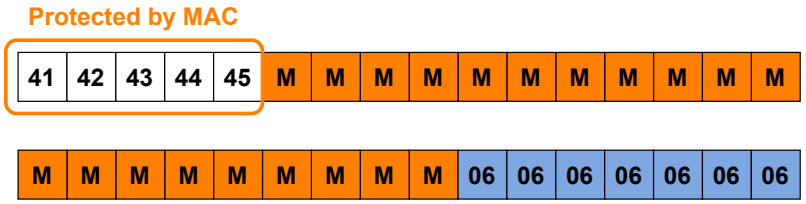


Figure 3.6.: When processing five plaintext bytes with AES-CBC and HMAC-SHA, the encryptor needs to append 20 bytes of the HMAC-SHA output and seven bytes of padding, each set to `0x06`. The MAC is computed over the implicit record sequence number, record header field, and plaintext data.

MAC is computed over the plaintext (fragment) and additional metadata like the sequence number, the type, and the payload length of the record.

$$\text{MAC}_k(m) := \text{HMAC}(SQN \,||\, Type \,||\, Version \,||\, \text{len}(Fragment) \,||\, Fragment)$$

TLS specifies the value of the padding bytes. The last byte of the padded plaintext specifies how many padding bytes are used, excluding that last byte. In TLS versions other than SSLv3, the value of the rest of the padding bytes is identical to the value of the last byte. For example, if four padding bytes are used (including the last byte), then the value of all four bytes will be `0x03`. In SSLv3, the remaining padding bytes had random values, resulting in a non-deterministic padding scheme.

To demonstrate the full process, if a peer encrypts five bytes of data with the `TLS_RSA_WITH_AES_256_CBC_SHA` cipher suite, it uses HMAC-SHA (whose output is 20 bytes long) and AES-CBC. After computing HMAC-SHA for the original plaintext, the concatenation is 25 bytes in length, which fits into two AES 16-byte blocks. The encryptor will typically select the minimum viable amount of padding, which would be seven bytes in this case. The first block contains the data and the first 11 HMAC bytes. The second block contains the remaining nine HMAC bytes and seven bytes of padding `0x06` (see Figure 3.6). Note that the peer can also choose longer padding and append 23, 39, ... or 247 padding bytes (while setting the value of the padding bytes accordingly).

TLS offers `BLOCK` ciphers for AES, DES, 3DES, IDEA, SEED, CAMELLIA and RC2.

Stream ciphers. For `STREAM` ciphers, the process is very similar to that of a BLOCK cipher, except that the padding step is omitted. The only available 'real' `STREAM` cipher is RC4.

AEAD Ciphers. `AEAD` ciphers were introduced in TLS 1.2 and are not allowed to be used in older versions. As their name implies, they use authenticated encryption algorithms, and therefore fix a design flaw of `BLOCK` ciphers (see 4.3).

TLS offers three different AEAD cipher suite groups:

- GCM
- CCM
- Chacha20 Poly1305

The additional authenticated data (AAD) for these ciphers is the concatenation of the sequence number, the record type, the record version, and the length of the plaintext. The IV for `AEAD` ciphers is constructed from both an explicit part as well as an implicit part. The implicit part is typically 4 bytes long and is taken from the key block, whereas the explicit part is chosen by the sender and send in front of the ciphertext with each record. The IV selection is left to the sender, which was a dangerous design decision, as inappropriate IV selection can lead to severe attacks (see Section 4.4).

Null Ciphers. Another group of special cipher suites are the so-called "null" ciphers. These cipher suites do not provide confidentiality, as they do not encrypt the data of the record layer and only provide authentication and integrity. Null cipher suites are generally considered insecure, as they do not offer all of the security goals TLS promises. Technically, null ciphers are `STREAM` ciphers.

Export Ciphers. Another special class of cipher suites are the export cipher suites. These cipher suites stem from the early days of cryptography, where strong cryptography was still considered a weapon by US export laws.[1] To still be able to use TLS outside of the US in restricted countries, special cipher suites were created with intentional weakened cryptography. In these cipher suites, public key algorithms do not exceed 512-bit of strength, and symmetric algorithms do not exceed 40-bit. Since using these cipher suites needs weakened public keys, the server transmits these in a `ServerKeyExchange` message, even when RSA is used. To realize 40-bit strong symmetric ciphers, the peers first extract 40 bits from the key block, and then use these 40 bits to compute the full length keys. This computation is done by applying the PRF with the label "client write key" and "server write key" respectively, as well as the client and server random (in this order) to the secret. The IVs in export ciphers are also generated differently. They are generated using the PRF with an empty secret, the label 'IV block' and then nonces. The IV block then contains the concatenation of the client IV and the server IV. Export cipher suites are generally considered insecure, as the used key lengths can be easily broken, even with consumer hardware.

Other Ciphers. As with key exchange algorithms, some more exotic, not entirely specified cipher suites exist. For example, the GOST cipher suites [56], also specify the CTR or CNT mode or WolfSSL which implements HC [57] and RABBIT [58] stream ciphers.

[1] `https://www.bis.doc.gov/index.php/policy-guidance/encryption/2-items-in-cat-5-part-2/a-5a002-a-and-5d002-c-1/ii-key-length`

3.9. MAC Algorithms

TLS offers MD5, SHA1, SHA256, and SHA384 as HMAC algorithms, where cipher suites that define SHA256 or SHA384 are officially only allowed to be used in TLS 1.2. If SHA384 cipher suites are used in TLS 1.2, SHA384 will also be used as the hash function in the PRF (see Subsection 3.6.1).

3.10. Public Key Infrastructure

The TLS protocol uses the Internet Public Key Infrastructure (PKI) to authenticate its peers. In a typical TLS scenario on the web, a server authenticates to a TLS client with an X.509 certificate [59] during the handshake. The certificate contains the server public key (used within the handshake), server domain name, expiration date, and several extensions. Extensions may, for example, define key usage (e.g., signing and encipherment), extended key usage (e.g., WWW server protection), or the location of the certificate revocation list. For the connection's security, the TLS client must carefully validate the certificate chain.

The certificate itself is signed by its issuer. A peer trying to validate a certificate has to use a path-building algorithm [60] to create a path to a trust anchor, which is typically a certificate authority that the peer trusts.

3.11. TLS Extensions

Since TLS 1.0, the TLS protocol introduced extension points into the `ClientHello` and `ServerHello` messages. The client can add extensions to its `ClientHello` to indicate support for additional features or to provide more information to the server. If the server implements the respective extension, it can use the information to guide its parameter selection process or to enable additional TLS features for the session by sending the according extension in its `ServerHello`. The server cannot send extensions in its `ServerHello` message that the client did not initially propose. The most important extensions are listed below.

Encrypt-Then-MAC. The *Encrypt-Then-MAC* extension [61] changes the way `BLOCK` ciphers encrypt data. The usual MAC-then-Pad-then-Encrypt scheme in TLS CBC was selected to intuitively provide an attacker with less information than a Pad-then-Encrypt-then-MAC scheme since the MAC is also encrypted and to make message-alteration attacks harder, as they would require breaking the encryption algorithm as well as the MAC. Unfortunately, for TLS further research revealed that this design allowed for a series of serious attacks [11,62,63] and is very hard to implement securely. Modern designs would use a PAD-then-Encrypt-then-MAC scheme (also known as Encrypt-then-Mac). This scheme allows a receiver of a record to first check the integrity and authenticity of a record before decrypting it. With this construction, it is less likely that information about the plaintext gets leaked from the decryption process. This is especially helpful with the CBC mode which is widely used in TLS.

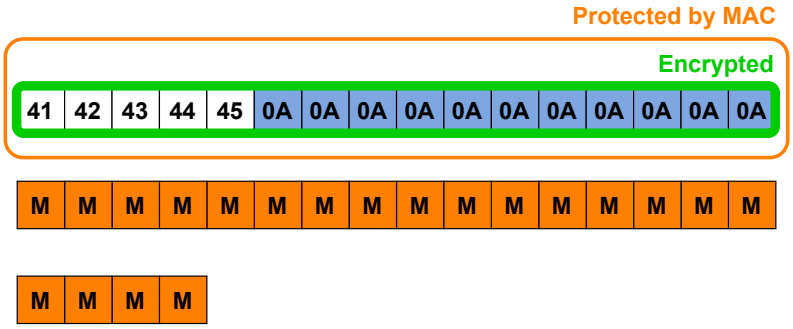


Figure 3.7.: When processing five plaintext bytes with AES-CBC and HMAC-SHA1 in Encrypt-then-MAC mode, the encryptor must append eleven padding bytes before encrypting the data. The MAC gets computed over the implicit record sequence number, record header field, and the ciphertext, resulting in additional 20 bytes.

If a client supports the Encrypt-then-MAC extension it indicates this by transmitting the (empty) Encrypt-Then-MAC extension in its `ClientHello` message. A server can choose to negotiate the extension by reflecting the extension in its `ServerHello` message. After that, both parties will not encrypt their MAC's anymore but will move to the Encrypt-then-MAC scheme as visualized in Figure 3.7.

Renegotiation Info. In 2009 a chosen prefix attack, called Renegotiation Attack, [64]) was discovered, which abused the renegotiation mechanism of TLS (see Section 3.13). A new extension was introduced to mitigate the attack to avoid issuing a new version of the protocol or disabling renegotiation on the server side. The client sends this extension in its `ClientHello`, and the server is supposed to reflect the extension to assure the client that it mitigated the issue. Then, if the peers decide to perform a renegotiation, both are supposed to send the renegotiation extension again. However, this time, both add the `verify_data` from the previous handshake to the extension to cryptographically tie both sessions together. The client only attaches its `verify_data`, while the server attaches a concatenation of both. As Section 4.9 will show, this binding was not strong enough.

Extended Master Secret. The *Extended Master Secret* extension was introduced to mitigate the *triple handshake attack* [65] (see Section 4.9). When this extension is negotiated, the MS computation is changed to include the transcript instead of only the client and server random. With the extension in place, any changes in the exchanged handshake messages do not only invalidate the `verify_data` in the `Finished` message but change the symmetric keys entirely.

Application Layer Protocol Negotiation. The *Application Layer Protocol Negotiation* (ALPN) extension [66] allows the TLS peers to select a specific appli-

cation layer protocol. With ALPN, a web server can offer different application layer protocols on the same port, for example, more performant versions of the HTTP protocol (in particular, HTTP/2 along with HTTP/1.1), while avoiding additional round trips for protocol negotiation. To agree on the application layer protocol, the client sends the protocols it supports as a list of strings to the server. The server selects a protocol it supports or sends an alert if no protocol supported by the server is found on the list. Protocol names are presented in ASCII and assigned by IANA.[2] The names for HTTP, IMAP, POP3, and FTP have already been standardized, while SMTP is not yet registered.

Server Name Indication. In some deployments, several web (or other) services may be hosted at the same IP address and port. To support this *virtual hosting* configuration, the client indicates the desired hostname in the *Server Name Indication* (SNI) extension [67]. The server can then use this information to choose an acceptable certificate. While SNI is well-supported by HTTPS servers, it is much less common in other application protocols such as SMTP, IMAP, POP3, and FTP.

3.12. Session Resumption

Asymmetric cryptographic operations are computationally expensive for all involved parties. The TLS protocol, therefore, introduced a feature called 'session resumption' to reuse already established key material and avoid expensive public key cryptography. TLS offers two session resumption mechanisms: session IDs and session tickets.

3.12.1. Session IDs

Session resumption via session IDs was already part of the protocol from the very beginning. Whenever a client connects to a server, the server assigns a session ID to the client in the `ServerHello`. If the session ends, both parties are supposed to store the session ID alongside the MS of the session for future connections. If a client then later reconnects to the server, it can transmit the session ID within its `ClientHello` if the client still remembers the MS of the previous connection. If the server also still remembers the tuple, it can respond with the same session ID in its `ServerHello`, indicating that it will perform a session resumption instead of a full handshake. The server then proceeds to send a `ChangeCipherSpec` and `Finished` message which the client also responds to with a `ChangeCipherSpec` and `Finished`. The resumption handshake is visualized in Figure 3.5a. The connection uses the same MS as the previous connection, but since the client and server random changed, different symmetric keys are used.

[2] `https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml#alpn-protocol-ids`

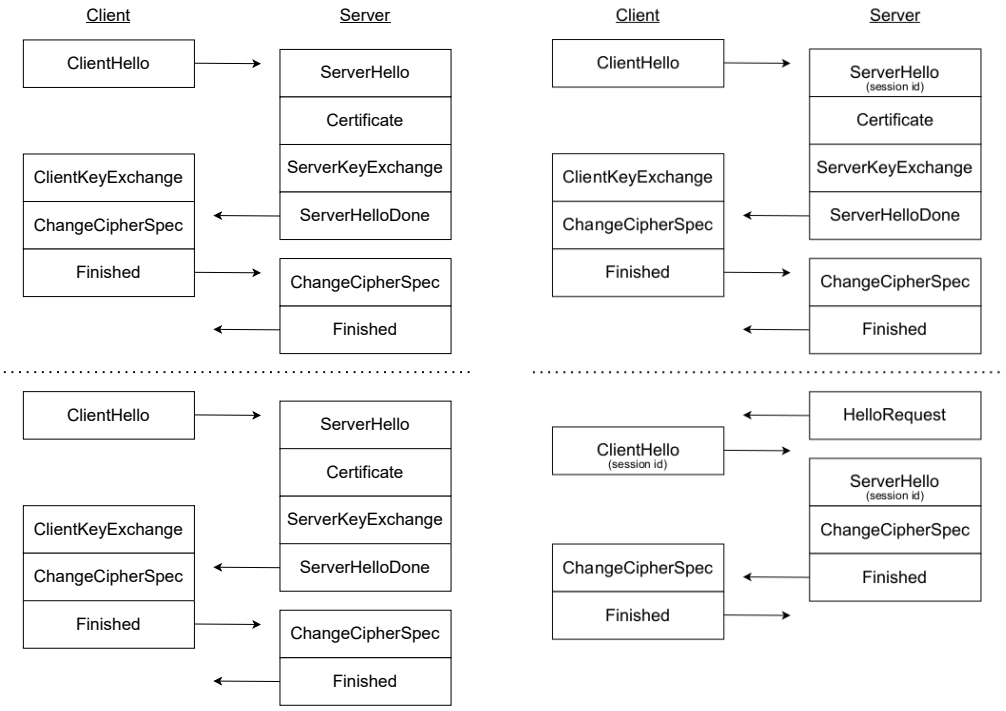
3.12.2. Session Tickets

The construction based on session IDs had a major drawback: the server had to store a session ID and MS for each client connection. For servers with many incoming connections, this could accumulate to a significant computational load on the server. Additionally, the session ID mechanism was unsuitable for CDNs where multiple servers are answering incoming client requests since the servers would need to synchronize the session ID cache to benefit from the performance boost. The solution to these problems was a new resumption mechanism called *session tickets* introduced as an extension in RFC 5077 [68]. With session tickets, the MS is only stored on the client machine. To perform a session resumption with session tickets, the client has to signal support for session tickets in its initial session by including a session ticket extension its `ClientHello`. The server then responds with the same extension to indicate that the extension will be active for the session. After the client sends its `Finished` message. The server now sends a `NewSessionTicket` message to the client, which contains a session ticket, e.g., the MS of the connection alongside other connection parameters, encrypted under a Session Ticket Encryption Key (STEK). The STEK is a symmetric encryption key that is only known to the server. The session ticket is usually also integrity protected. The client has to store the session ticket alongside the plaintext MS. The handshake then finishes as usual. If a client wishes to resume the session later, it can send the session ticket in the session ticket extension in the `ClientHello`. If the server still has the STEK for the ticket, it can decrypt the ticket to get the original MS. From there on, the session can continue analog to the resumption with session IDs.

3.13. Renegotiation

TLS offers the possibility to renew the keys and potentially renegotiate the parameters of the connection. To do so, either peer can initiate a renegotiation. The client can send a `ClientHello` message and if the server wishes to accept the renegotiation request, it can simply answer with a `ServerHello` message and the handshake continues as usual. The new handshake is encrypted under the currently exchanged connection parameters. Once the `ChangeCipherSpec` message of the new handshake is exchanged, the connection switches to the newly negotiated parameters. The server can send an appropriate alert message if it wants to decline the renegotiation.

Servers can also initiate a renegotiation by sending the client a `HelloRequest`. If the client wishes to accept the renegotiation, it can simply proceed with a `ClientHello` message, while if it wishes to decline, it can send an appropriate alert message. From there on, the connection continues as usual.



(a) Client initiated renegotiation.

(b) Server initiated renegotiation with resumption.

Figure 3.8.: Renegotiation mechanisms in TLS. Note that each renegotiation handshake could potentially use any regular TLS feature like client authentication or session resumption.

3.14. Client Authentication

In a usual TLS connection, only the server authenticates towards the client, but TLS also allows the client to authenticate towards the server. If client authentication is desired, it must be configured on the server. If the server wishes to request client authentication, it sends a `CertificateRequest` to the client before its `ServerHelloDone` message. The `CertificateRequest` message contains information to guide the client into choosing the correct certificate and a list of supported signature and hash algorithms. After the `ServerHelloDone` message, the client must send its certificate in a certificate message. The client then sends its `ClientKeyExchange` message followed by a `CertificateVerify` message, which signs the transcript. The signature proves to the server that the client possesses the private key for the provided certificate and that the client also wishes to authenticate in this particular session. If the client does not have a fitting certificate, it can send an empty `Certificate` message and ommit the `CertificateVerify` message to indicate its missing a certificate. The server can then see that the client is unable to authenticate. Depending on the application, the server may still allow the client to complete the handshake but will treat the client as unauthenticated. If the client certificate contains a fitting DH public key or ECDH public key for the parameters selected from the

server, the public key from the certificate will be used for the key exchange, and the `ClientKeyExchange` message will be empty.

3.15. TLS 1.3

With the release of TLS 1.3 [53] in 2018, the TLS protocol received a significant overhaul. The handshake protocol was fundamentally changed to improve the performance of the protocol and the security of the cryptographic computations. The design of TLS 1.3 was done through public discourse, with input from academia as well as from the industry. The cryptographic components of the TLS 1.3 handshake were proven to be secure by Dowling et al. [69] and many insecure features and algorithms of the previous versions were removed, like the CBC mode or the RSA key exchange. Additionally, the TLS 1.3 cipher suite design was overhauled, and all previous cipher suites were obsoleted. In TLS 1.3, cipher suites only define a bulk encryption algorithm and a hash function. The key exchange and authorization algorithms were decoupled from the cipher suites. As for key exchange algorithms, TLS 1.3 only allows DHE and ECDHE. The TLS 1.3 handshake is visualized in Figure 3.9. As with the previous versions, the TLS 1.3 handshake starts with the client and server exchanging hello messages. The most crucial difference is that the client already sends its public key for the key exchange within a *key share* extension. If the server supports the cryptographic group, it can directly respond with its public key in the `ServerHello`. This allows both parties to compute a shared secret directly after the `ServerHello`, which allows them to encrypt all messages after the `ServerHello`. The server follows the `ServerHello` with an `EncryptedExtensions` message that allows the server to send extensions that should be already encrypted. After that, the server sends its `Certificate` message, which, as before, contains the server's certificates. After that, the server sends a `CertificateVerify` message, which contains a signature with the certificate's private key that signs the transcript. Note that up to this point, the key share of the server was not authenticated. The server then finishes its flight with a `Finished` message, which contains the `verify_data` of the session. The client then finishes the handshake by sending its `Finished` message. Suppose the client did not correctly guess the server-supported named groups in its initial key share extension. In that case, the server responds by sending a `HelloRetryRequest`, disguised as a `ServerHello` message, indicating which groups the server supports. The client then sends another `ClientHello` containing a usable key share. The handshake then continues as usual.

3.15.1. Certificate Verify Signatures

The signatures in TLS 1.3 are specifically designed to be incompatible with the signatures generated by previous versions. To achieve this goal, the content that is supposed to be signed is prepended by 64 times `0x20`, a label, and a separation byte (`0x00`). The first 64 bytes ensure that no signature can be confused with a

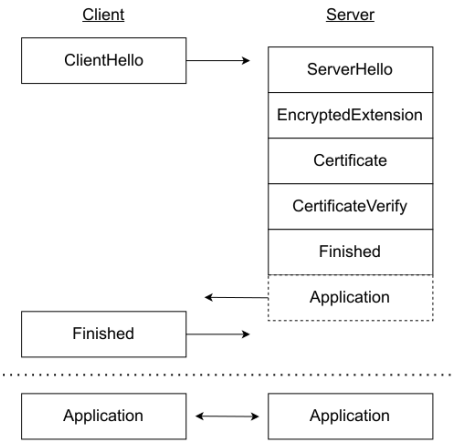


Figure 3.9.: A TLS 1.3 handshake without compatibility mode.

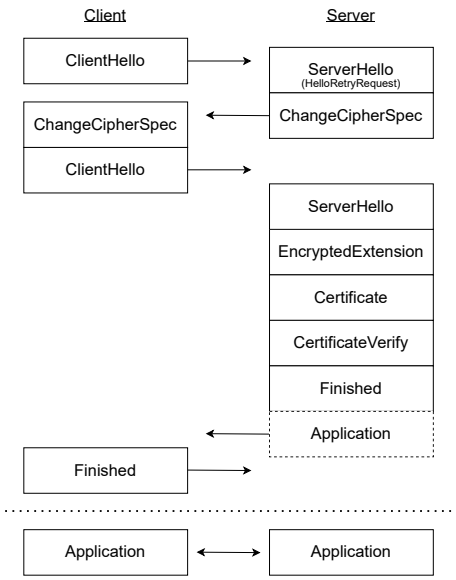


Figure 3.10.: A TLS 1.3 handshake with compatibility mode and `HelloRetryRequest`.

`ServerKeyExchange` from previous versions, as previous versions would sign the client and server random at these byte positions, which will never realistically contain `0x20` if at least one party is honest.

3.15.2. Key Derivation

A major shift in TLS 1.3 is the key derivation. TLS 1.3 removes the previous keyblock-based approach and replaces it with a modern HKDF construction. The key derivation is visualized in Figure 3.11. The early traffic secrets keys protect 0-RTT data (early data), the handshake traffic secret keys protect the handshake traffic after the `ServerHello` message, and the application traffic secret keys protect the transmitted `Application` data after the handshake.

3.15.3. Record Layer

The record layer of TLS 1.3 was revised. `BLOCK` and `STREAM` ciphers were removed, which leaves only `AEAD` ciphers. Once active, the record type in each record is set to *application data*, while the true record type is now encrypted and transmitted behind the plaintext. Each record can also contain an arbitrary amount of padding bytes (always `0x00`) transmitted after the record type. This padding is optional and can be used to hide the length of the plaintext. The additional authentication data (AAD) of each record is the record header, concatenated with the sequence number.

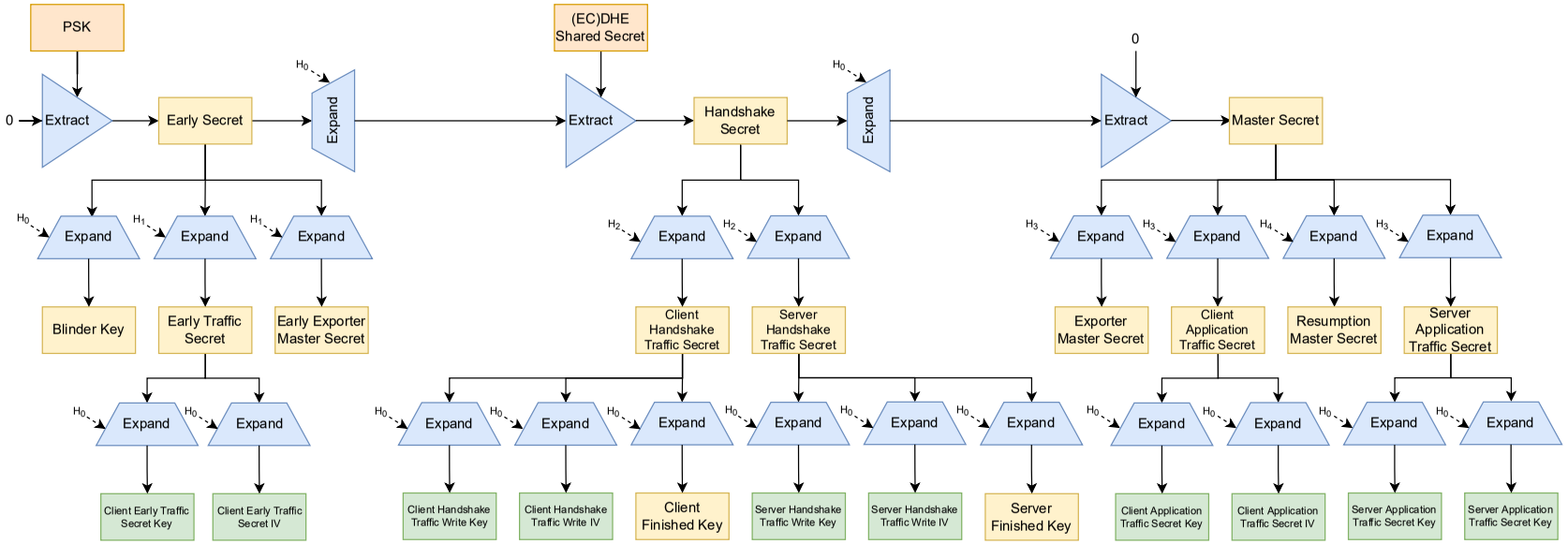


Figure 3.11.: The key derivation of TLS 1.3. `0` indicates an empty byte array. Inputs entering the extract functions on the flat side are used as the salt, while inputs from the angled side are used as the key. Inputs entering the expand function at an angle are transcript hashes, while inputs entering on the flat side are keys. H_0 refers to an empty transcript; H_1 refers to the transcript after the **`ClientHello`**; H_2 refers to the transcript after the **`ServerHello`**; H_3 refers to the transcript after the server **`Finished`** and H_4 refers to the transcript after the client **`Finished`**.

3.15.4. Backwards Compatibility

During the specification process of TLS 1.3, the protocol was also already implemented to perform real-world experiments. During these experiments, it was shown that some of the implemented changes would interfere with currently deployed implementations which did not follow the original TLS specification well and would break connections. To mitigate this issue and to be able to deploy TLS 1.3 immediately, the protocol design was changed to avoid these issues.

Protocol Version Negotiation. A major step towards compatibility were changes to the version negotiation. Setting the highest client advertised version in the `ClientHello` to TLS 1.3 revealed that some servers could not negotiate TLS 1.2 anymore, as they suffered from protocol version intolerance. To mitigate the issue, the protocol version field in the hello messages were deprecated and frozen to TLS 1.2. The actual protocol negotiation is done in a newly introduced extension called *protocol version*. In this extension, the client communicates all its supported versions. The server can then use this information to select the highest protocol version both peers support and communicate its choice in a protocol version extension within the `ServerHello`.

Hello Retry Request. `HelloRetryRequest` messages did not receive their own handshake message type but instead use the message structure and type of the `ServerHello` message. This design choice was done since some TLS 1.2 middlebox implementations could not process messages containing newly introduced handshake message types. To indicate that the transmitted message is actually not a `ServerHello` message but a `HelloRetryRequest` message, the message contains a hard-coded random value (SHA256("HelloRetryRequest")). If a client receives such a `ServerHello`, it will see the hard-coded random value and treat it as a `HelloRetryRequest` instead.

Record Layer Version. The record layer version was deprecated. The record version of the `ClientHello` should be TLS 1.0, and all further records are transmitted with the protocol version TLS 1.2.

3.15.5. 0-RTT

TLS 1.3 introduced a new feature called 0-RTT. The idea behind 0-RTT is to transmit encrypted application data directly after the `ClientHello` message before waiting for a response from the server. The 0-RTT mechanism in TLS 1.3 can only be used when a previous session has already been established with the server. The 0-RTT data is then protected with a secret derived from the previous session. The server can then decide if it wants to accept the early data or not. It can send an *early data* extension to the client if it wants to accept the data. Once the client has received the server flight, it can send an `EndOfEarlyData` message to indicate that it will continue with the handshake.

This design forces a server that does not possess the ability to process the 0-RTT data, to try to decrypt all 0-RTT data (and fail). Once it finds a de-

cryptable record, the server discovers the `EndOfEarlyData` message and the handshake proceeds. The 0-RTT design has another major disadvantage: the 0-RTT data is not protected from replay attacks, e.g., a server cannot know if the same `ClientHello` with the same 0-RTT data was processed before without spending tremendous resources on storing these values. This can be tough, especially in distributed systems like CDN applications where multiple servers would have to share the same database.

3.16. STARTTLS

Unprotected legacy protocols can be extended to support TLS by upgrading a plaintext connection using a protocol-specific STARTTLS command. After that command, the client switches to TLS and starts a regular TLS handshake as if it had just established the TCP connection. After the TLS handshake succeeds, the application protocol continues, where the TLS record layer now protects each message. STARTTLS was first standardized in RFC 2487 [70] as an extension to SMTP.

3.17. False Start

While not part of the original TLS 1.2 specification, TLS also allows the client to send application data with its `Finished` message. This behavior is called *False Start* and is specified in RFC 7918 [71]. False start effectively saves one round trip time and improves the session's performance. The false start feature shall only be used with 'secure' cipher suites with secure versions. RFC 7918 [71] makes recommendations about what connections are considered secure, but ultimately each implementations is free to decide upon itself.

3.18. DTLS

While TLS was designed to use TCP as a transport layer, some applications can benefit from an unreliable transport layer like UDP. For this purpose, DTLS was created. DTLS is directly derived from TLS and contains changes to enable a secure and robust handshake. For this purpose, DTLS introduces three major changes.

Record Layer. The first major change was made in the record layer. First, the cipher type `STREAM` cannot be used within DTLS, as stream ciphers cannot reasonably operate if parts of plaintext are missing. The second major change is that the sequence number of each record, which was only implicitly tracked in TLS, is now explicitly added to each record. This way, even if a record gets lost in transmission, the sequence number of the record will still be known to the peer, enabling the peer to decrypt the record. Additionally, each record has an epoch number attached to it. The epoch number is an identifier of the keys used to encrypt the record. The handshake starts with epoch zero and is increased to

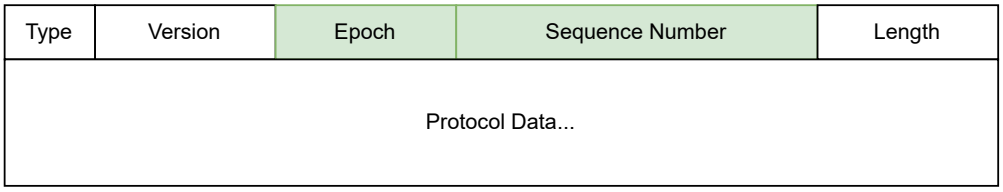


Figure 3.12.: A DTLS record header.

epoch one after the `ChangeCipherSpec` message. Each following renegotiation increases the epoch by one. With this information, each record can be decrypted independently. The DTLS record header is visualized in Figure 3.12.

Fragmentation. DTLS introduced additional fields into all handshake messages (message sequence number, fragment offset, fragment length) such that fragmented handshake messages could be reconstructed (see Figure 3.13). The message sequence number indicates to which message the fragment belongs. The fragment offset indicates at which position the given fragment has to be inserted, while the fragment length indicates how long the fragment is.

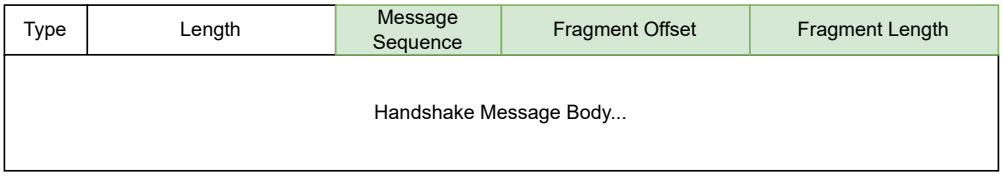


Figure 3.13.: A DTLS handshake message header.

Retransmissions. Since DTLS messages can always get lost in transit, DTLS needs a retransmission mechanism to function correctly with an unreliable transport layer. For this purpose, DTLS retransmits the whole message flight if a peer did not send a complete answer within a given timeframe. Note that the retransmission is not a one-to-one copy but is transmitted in a new record with a new record sequence number and re-encrypted if the record layer is already active.

Denial of Service Protection. Due to the nature of UDP and IP, it is possible to spoof the source IP address of a UDP packet. If an attacker sent a spoofed DTLS `ClientHello` to a target server, the server would respond to the spoofed IP address. An attacker could use the DTLS server to disguise the origin of DoS attacks and even as an amplifier to attack other servers. To avoid sending big messages due to the server's certificates in the first flight (acting as an amplifier), DTLS introduces an additional roundtrip time that the server can optionally implement. Upon receiving the first `ClientHello` message, the server sends a `HelloVerifyRequest` message, which contains a cookie that is derived from the `ClientHello`, the client IP, client port, and a server known secret. The client then transmits this cookie in a `ClientHello` message that is (besides the cookie)

identical to the first `ClientHello` message. The server can then recompute the cookie and check that it matches the cookie sent to the client. If it does, the server knows that the peer was able to receive messages from the server, and the `ClientHello` was not created by a spoofing attack.

4

Attacks on TLS

Since the creation of SSL/TLS in 1994, the protocol has been subject to many different attacks. A brief timeline with the most important attacks on the protocol is presented in Figure 4.1. Most notably, the timeline is comparably empty between the early 2000s and 2011. Until then, mostly MitM attackers were considered when analyzing TLS, and it was assumed that the protocol was reasonably secure. This changed when Thai Duong and Juliano Rizzo published their BEAST attack [25] in 2011. The BEAST attack introduced the MitB attacker model which enabled a variety of new attacks on the protocol, which were previously only considered theoretical issues. Along with the increasing importance of TLS, this created immense academic interest in the protocol which resulted in lots of new attacks on the protocol. This *golden age* of TLS attacks lasted until roughly 2016.

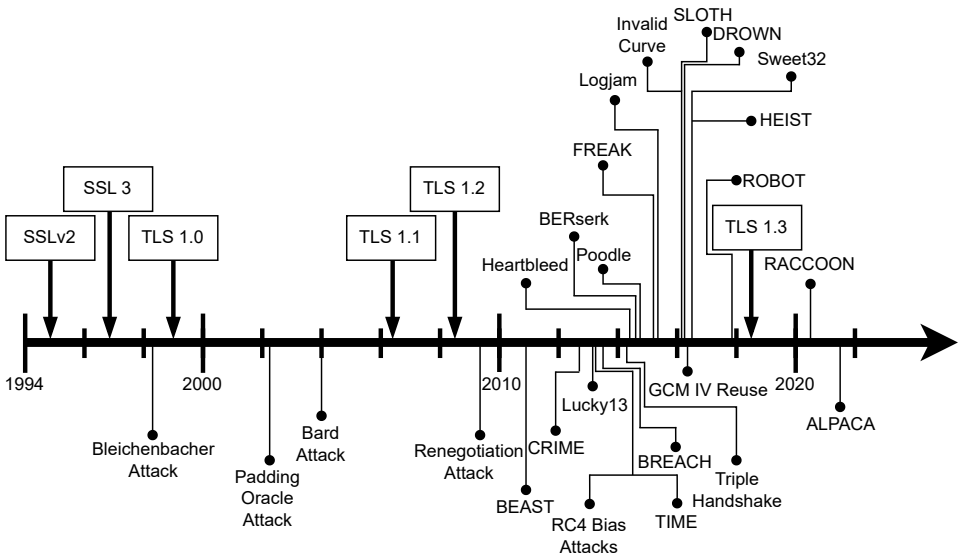


Figure 4.1.: A sketch of the timeline of the most prominent attacks on the TLS protocol.

Since then, research around the protocol has cooled off, and the focus of the community has shifted from novel attacks on the protocol to security proofs, exploitation of implementation flaws and analysis of the ecosystem. This chapter introduces the most prominent attacks on the protocol, that were already known before my contributions to the field.

4.1. BEAST Attack

The BEAST attack is the logical successor of the Bard attack [72,73]. The Bard attack is cryptographically identical to the BEAST attack, but the Bard attack did not consider a MitB attacker. Omitting the history of the attack, only the BEAST attack shall be introduced here.

The BEAST attack, published in 2011, introduced a working CPA attack that was functional in some browsers at the time, against TLS 1.0 in CBC mode. The attack was the first attack of its kind and introduced the MitB attacker model to TLS, which ultimately allowed the attacker to send chosen plaintext within a TLS connection. The attack requires MitM and MitB capabilities and a TLS 1.0 connection from the victim to the target server with a CBC cipher suite. The attack works as follows: First, the attacker lures the victim onto the attacker's website. From there, the attacker can initiate HTTP requests to any other site from the victim's browser. Due to the same origin policy, the attacker can not access the cookie of the other website directly. However, since the attacker can control parts of the HTTP request, it can influence where the cookie gets placed within a ciphertext block, for example, by carefully selecting the length of the URL. As the structure of an HTTP request is known, the attacker can construct requests where a ciphertext block contains known bytes as well as the first byte of the cookie (that the attacker does not know yet), with the remaining cookie bytes spread across subsequent blocks. The attacker then carefully observes the ciphertext of the request. The attacker can then force the victim to send a request with the plaintext $m_i = C_{i-1} \oplus m_j \oplus m_{j-1}$, where j is the index of the block that contained the first cookie byte and C_{i-1} is the last ciphertext block of the previous record. Since the attacker does not know the last byte of m_j, the attacker guesses the value for the last byte. By constructing the plaintext in this way, the input to the block cipher would be $z_i = C_{i-1} \oplus m_j \oplus m_{j-1} \oplus C_{i-1} = m_j \oplus m_{j-1} = z_j$. If the attacker guessed correctly, the previously mentioned equation holds and the attacker would notice that its ciphertext block is identical to the ciphertext block of the targeted plaintext block. If the attacker did not guess correctly, the attacker simply tries again with a different value for the guessed plaintext, until the attacker is successful. After at most 255 tries, the attacker knows the previously unknown plaintext byte. Once the attacker learned the first byte they can adjust the query to shift the plaintext by a single byte such that the target block will now again contain a single unknown plaintext byte again. This way, the attacker is able to learn the entire cookie value byte-by-byte.

The previous description of the attack omitted the detail that the attacker actually has to be able to choose arbitrary plaintext at the beginning of a record

of a new flight. As the attack relies on HTTP requests, header bytes are always placed before any payload data. This is a problem for the attacker as they only know the IV used for the first block of the next record and this first block is not fully under their control. Consequently, the attacker is not able to correctly prepare the plaintext.

Duong and Rizzo circumvented this limitation with different browser features and web technologies [25]. They called a feature that allows them to circumvent this limitation an *agent*. The authors were able to build such an agent with Java applets, Microsoft Silverlight, and an older version of HTML5 WebSockets. The TLS protocol is supposed to protect the data even in the presence of such agents. It is plausible that agents are also present in modern browsers; however, modern TLS implementations as seen in current browsers implement a non-breaking but not officially standardized countermeasure.

BEAST Mitigation. To harden TLS 1.0 implementations against the block-wise adaptive attacks, Möller proposed a countermeasure for SSL and TLS implementations in 2002 on the TLS Mailing [74] list, which would mitigate the mentioned attacks [25, 72, 73] without changing the protocol itself. This countermeasure is known as the *0/n split*. Implementations should send an empty record before each actual data transmission. The empty record consists only of the encrypted MAC and padding. This means that whenever a TLS API user would call the `send()` function, the TLS implementation would send an empty record and then the normal TLS records to transmit the actual data. The empty record consists only of the encrypted MAC and padding. The IV of the empty record is still predictable by an attacker, but since the whole record does not allow for the transmission of chosen plaintext, the IV for the subsequent record, which then transmits `Application` data is unpredictable. The IV is unpredictable since the increasing record sequence numbers for the MAC computation change the MAC which then becomes unpredictable by the attacker. Therefore, the ciphertext of the (unpredictable) MAC cannot be predicted either. Since the last block of the empty record will be used as an IV in the actual data record (containing potentially chosen plaintext), the IV is not predictable anymore and therefore not exploitable by the Bard/BEAST attack.

The *0/n split* was implemented in OpenSSL 0.9.6d in May 2002. However, it was reported that some (buggy) implementations broke the connection if they received empty TLS records [75]. This forced the developers to implement the *1/n-1 split*. In this version of the mitigation, instead of using a completely empty record, only a single byte of the plaintext gets transmitted (the rest of the record only contains the MAC and padding). The rest of the `Application` gets transmitted in the next record. This one byte can be chosen plaintext, but since the other bytes of the record still contain the MAC and padding, the IV for the rest of the plaintext is still not predictable. The one byte with the predictable IV offers not enough control to the attacker to perform the attack.

4.2. Compression Oracles

CRIME. Duong and Rizzo released another attack on TLS and SPDY, merely one year after the BEAST attack, which they named CRIME [76] (**C**ompression **r**atio **I**nfo-Leak **m**ass **e**xploitation). The CRIME attack, like the BEAST attack, works in a combined MitM and MitB attacker model and abuses the build-in compression of the TLS connection. The core idea is that if a secret, e.g. the cookie of connection looks similar to other (attacker controllable) plaintext, the overall size of ciphertext decreases. If an attacker can partially control enough parts of the plaintext, the attacker can continuously try to make its plaintexts look more and more like the secret. Whenever the attacker succeeds, the ciphertext is compressed a little bit more than if the attacker guessed wrongly. The CRIME attack was demonstrated by abusing RC4 but also works with AEAD or BLOCK ciphers, with the latter requiring some additional engineering.

TIME. At the Black Hat Conference in 2013, the TIME attack [77] was introduced, which contained a new technique that allowed an attacker to perform the CRIME attack without MitM privileges. An attacker can carefully control the amount of content that is put within an HTTP request, such that if the plaintext gets compressed more (e.g, the attacker made a correct guess), the data of the request fits into a single TCP frame, while if the plaintext did not compress as much, two distinct TCP frames were needed. This could cause a notable delay in the reception of the data, which could be measured by the attacker. This effectively converted the CRIME attack into a timing attack with a less privileged attacker.

BREACH. Another related compression attack is the BREACH attack [78]. While the CRIME attack targeted the build-in TLS compression, the BREACH attack used the compression that was added to the HTTP protocol. HTTP compression does not compress the cookie, which means that BREACH cannot steal cookies (as they are transmitted as part of the header); instead, BREACH targets the payload of an HTTP connection, which for example contains CSRF tokens.

HEIST. The HEIST attack was published in 2016 [79] and is another extension of the CRIME attack. The attack is exploring similar ideas to the TIME attack [77], but also explores the `fetch()` API and HTTP 2.0, which did not exist when the original attack was published.

4.3. CBC Padding Oracle Attacks

One of the main design failures in SSLv3 and TLS is the specification of the *MAC-then-Pad-then-Encrypt* scheme in CBC cipher suites. This scheme was responsible for a series of attacks on TLS implementations named padding oracle attacks. Even though the countermeasures are explicitly summarized in the TLS

specification [52, Section 6.2.3.2], their correct implementation is challenging.[1]

4.3.1. Vaudenay's Padding Oracle Attack

In 2002, Vaudenay showed that the MAC-then-Pad-then-Encrypt scheme introduces potential vulnerabilities in security protocols, in the form of so-called padding oracles [62]. The padding oracle attack exploits the malleability of the CBC mode. In this work, we focus on the case of TLS.

Consider the TLS record layer when using CBC mode. After decryption, the decrypting party must verify the padding and MAC bytes. The natural way to implement these two checks is first to verify the padding bytes and, if they verify correctly, then verify the MAC bytes. If the padding bytes are invalid, it is natural for an implementation to emit an error message, without checking the MAC bytes. On the other hand, if the padding bytes are valid but the MAC is invalid, it is then natural to emit a (potentially different) error message.

Assume a decryptor that indeed emits two different error messages in these cases. The attacker can decrypt the last byte of any message block p_i as follows. He sets the last ciphertext block to c_i and replaces the last byte of the previous block c_{i-1} with a value between 0 and 255. If after decryption the last cleartext byte is `0x00`, then the padding will be valid (other forms of valid padding are much less likely). When the padding byte correctly verifies, the attacker detects this by observing that the decryptor emitted an 'invalid MAC' error, rather than an 'invalid padding' error. Once the attacker observed an 'invalid MAC' error, the attacker learns the last byte of p_i by computing $p_i = c'_{i-1} \oplus c_i \oplus 0x00$. Using his knowledge of the last plaintext byte, the attacker can proceed to decrypt the second-to-last byte of p_i by shifting the plaintext like in the BEAST attack.

Note that the above attack relies on the ability to distinguish between ciphertexts decrypting to valid and invalid padding. It would therefore appear trivial for TLS implementations to prevent this attack by making sure they always emit the same error message. Indeed, Vaudenay was unaware of a way for an attacker to directly distinguish between these two cases in the context of TLS. The reason is that even if the TLS error messages differ, their distinction is impossible since they are encrypted with TLS session keys.

4.3.2. POODLE

SSLv3 uses the same *MAC-then-Pad-then-Encrypt* scheme, as TLS, but unlike TLS, the value of the padding bytes in SSLv3 are flexible. The last byte of the plaintext denotes how many padding bytes are present, but the rest of the padding bytes can take any value.

Consider a message with one full block of 16 padding bytes and an AES-CBC cipher suite. The last block of plaintext will have the value `0x0F`, and the first 15 bytes can take any value. Therefore, an attacker can replace the last block with any other ciphertext block c_j. The attacker then has a 1/256 chance that this modification will result in the last byte to be decrypted to `0x0F`, which

[1] We note that the countermeasures summarized in RFC 5246 [52] do not protect from timing-based attacks [11].

is a valid padding. Since the attacker did not modify the ciphertext of the MAC (and the MAC does not cover the padding), the peer will not notice the attacker's modification. If this happens, the attacker learns the last byte of m_j. This attack is known under the name 'POODLE' [63]. Although by the time of its discovery TLS was already used in much newer versions, the attack was devastating for many protocol implementations. The reason was that browsers were performing a so-called *downgrade-dance*. Whenever a browser failed to connect to a server it would try again with a lower protocol version until the browser had success. An attacker could abuse this feature to downgrade any version to SSLv3 to perform the POODLE attack.

Although POODLE relies on the flexibility of the padding bytes in SSLv3, it surprisingly also affects TLS implementations. In essence, there is nothing forcing a careless TLS developer to verify the (specified) padding bytes after decryption; a TLS implementation will interoperate just fine even if it does not check the padding bytes at all. In fact, it is *easier* for the developer to reuse the same code that handles SSLv3 padding in a TLS implementation. This has led to a variant of the POODLE attack that affects TLS implementations [80] called TLS-POODLE. Even after these two high-profile discoveries, variants of POODLE continued emerging [81–83]. These works detected different TLS record processing vulnerabilities; some TLS implementations only verified the first MAC byte, while others skipped validation of specific padding bytes.

4.3.3. Lucky 13

In 2013, AlFardan and Paterson [11] used a similar technique to break TLS confidentiality and dubbed their attack 'Lucky 13'. The attack relies on an important observation: The HMAC function requires different processing times when processing inputs of different lengths. By performing clever padding byte manipulations, the attacker can force the server to execute HMAC computations on plaintexts of different lengths. This is because the padding length determines the amount of data used as input into the HMAC function. The attacker can then measure the different processing times and learn information about the padding byte. We refer the reader to [11] for the full attack description.

The mitigation for Lucky 13 was to change the MAC verification code in TLS implementations to be constant-time, regardless of the number of processed cleartext blocks. This is possible, but writing and maintaining such code is hard, even for experts.

Amazon's s2n TLS library was released in 2015 [84], after the Lucky 13 attack was published. s2n's developers were aware of Lucky 13 and introduced specific countermeasures that seemed to render the code constant-time, thereby preventing the attack. They also introduced randomized timing delays to make the attack more difficult, in the unexpected case that the code turned out to be vulnerable. Despite all these efforts, s2n was still vulnerable to variants of Lucky 13 [85, 86].

4.3.4. Other Padding Oracles

Besides the already mentioned padding oracle vulnerabilities in TLS [80,82,83], there exist multiple other variations of them. These padding oracles sometimes manifest only in very specific and narrow circumstances, like off-by-one errors. A particularly interesting case is CVE-2016-2107 [87], which used records without a MAC to trigger the vulnerability.

4.4. GCM Nonce Reuse

While AEAD ciphers fixed the issues of CBC ciphers in TLS, the way AEAD ciphers were introduced left room for another class of attacks. If the GCM mode is used, the IV of a record should never be repeated. A repetition would allow an attacker to compute the authentication key, which would in return allow the attacker to forge arbitrary messages. The attack was analyzed in the context of TLS by Böck et al [88].

4.5. Bleichenbacher's Attack

Bleichenbacher's attack [12] is also a form of a padding oracle attack. Rather than targeting symmetric encryption, it targets a padding scheme used in RSA encryption called PKCS#1 v1.5. It exploits the malleability property of RSA encryption and relies on a decryptor (i.e., a server) acting as an oracle for (in)correctly padded cleartexts. The standard countermeasure is similar to that of CBC padding oracles; the server must not behave differently when encountering error states in RSA decryption. This countermeasure has become part of the TLS standard, but as with CBC padding oracles, secure implementation has proven challenging.

4.5.1. ROBOT

In 2018, Böck et al. performed a study on Bleichenbacher's attack on TLS in the wild called ROBOT [89]. They found vulnerabilities in servers used by high-profile websites such as Facebook and Paypal. Interestingly, their vulnerabilities could be triggered by using different TLS protocol flows or exploiting TCP connection states (TCP resets or timeouts). As with CBC padding oracles, Bleichenbacher's attack reappears every few years in the TLS ecosystem within a different context [90–92].

4.5.2. DROWN

The DROWN attack [93] is a Bleichenbacher attack on SSLv2 that can also be used to attack newer protocol versions. When DROWN was introduced in 2016, it was common that SSLv2 was still supported, but effectively not used, or had all cipher suites deactivated. The DROWN attack has huge impact on the ecosystem, as a bug in OpenSSL allowed using SSLv2, even though it was

seemingly disabled (by deactivation of all cipher suites). As SSLv2 was not thoroughly introduced, we refer to Aviram et al. [93] for details on the attack.

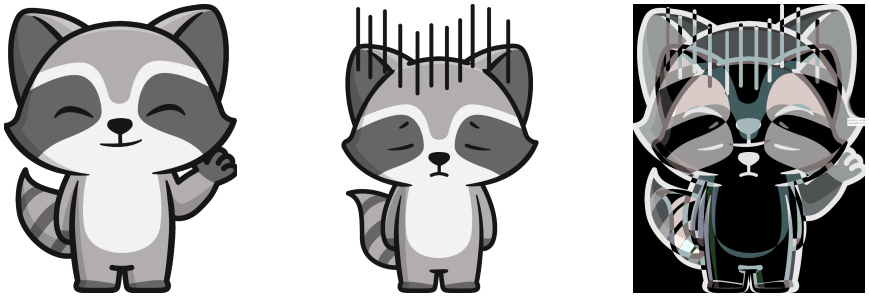
4.6. Sweet 32

Block ciphers consist of two important parameters: the key length and the block size. Since 2015, the NIST recommends a key length of at least 112 bits for symmetric ciphers [94]. But even if the key length is long, if the block size of a cipher is too small, collisions may occur once a certain number of blocks have been transmitted. These collisions are known to reduce the security of standardized modes of operation, such as CBC and CFB [95]. Hence, the amount of data encrypted without changing the keys should be limited for these ciphers.

Modern block ciphers usually use 128-bit blocks, while legacy block ciphers like DES, 3DES, or IDEA have 64-bit blocks. The block size plays a huge role in the determination of how many blocks can be encrypted under a given key. For example in CBC mode, due to the birthday paradox, after $2^{n/2}$ blocks a ciphertext collision has happened with a probability of 50%. If such a collision is found, the attacker can learn useful information about the plaintext, without knowing the secret key. For the CBC mode in particular, if an attacker ever observes two identical CBC-encrypted blocks (say C_i and C_j), they immediately know that both blocks were generated by the same input in the block cipher (since block ciphers are permutations). The attacker, therefore, knows that: $C_{i-1} \oplus X_i = C_{j-1} \oplus X_j$. By rearranging the equation the attacker learns that: $C_{i-1} \oplus C_{j-1} = X_i \oplus X_j$. C_{i-1} and C_{j-1} are publicly know, therefore the attacker learns the XOR of the two plaintexts X_i and X_j. This extracted information is visualized in Figure 4.6. While it is clearly possible to extract *some* information from this, most real-world applications are not broken by this attack alone. However, if an attacker ever finds a collision between a *known* plaintext and an unknown plaintext, they can immediately solve the whole equation and learn the missing plaintext i.e. if the attacker is in possession of either image in Figure 4.6, it can compute the other image.

In 2016, Bargahvan and Leurent evaluated the impact of 64-bit block ciphers in TLS, SSH, and IPSEC [96] and called their attack Sweet32. The shown attack requires the same attack scenario as in BEAST. The attacker acts a MitB and MitM and forces the victim to send a lot of HTTPS requests over a single connection (each containing a cookie). Since the attacker usually knows the content of the HTTPS request but the cookie, they can wait for a collision between a block of known plaintext and a cookie block. Once a collision appears, the attacker can immediately compute a block of the cookie. Bargahvan and Leurent were able to show that this attack can be used to steal a 32-byte HTTPS cookie with the transmission of 785 GB of data within 38 hours in their setting.

Note that the Sweet32 attack has several limitations. The attack requires that the TLS connection stays connected for the transmission of hundreds of GB of data. Depending on the application, this might be infeasible. Additionally, it requires that the application repeatedly transmit the secret (i.e., the cookie). If



(a) Plaintext A (b) Plaintext B (c) $A \oplus B$

Figure 4.2.: Visualization of block collisions in the Sweet32 attack. If an attacker ever observes two identical ciphertext blocks C_i and C_j, the attacker can compute the XOR of the plaintexts by computing $C_{i-1} \oplus C_{j-1} = X_i \oplus X_j$. If the attacker either knows X_i or X_j, it can compute the other.

the application only transmits the unknown plaintext once, the Sweet32 attack cannot be realistically applied.

4.7. Logjam

The Logjam attack is an attack on finite field Diffie-Hellman with export cipher suites. Assume a MitM attacker and a server supporting `DHE_EXPORT` cipher suites, while a connecting client is configured to only support `DHE` cipher suites. The MitM attacker can now change the cipher suites advertised by the connecting client to seem like the client only supports `DHE_EXPORT` cipher suites, leaving the server no other choice but to choose an `DHE_EXPORT` cipher suite. The server, therefore, creates a `DHE_EXPORT` `ServerKeyExchange` message and sends it (together with its `ServerHello` and `Certificate`) to the client. The MitM attacker then changes the cipher suite in the `ServerHello` to a regular DHE cipher suite. Depending on the minimum weakest acceptable parameters of the client, the client may accept the export parameters as normal parameters, as the `ServerKeyExchange` messages are otherwise indistinguishable. The attacker can then break the connection by solving the discrete logarithm problem, as the connection uses only 512-bit parameters.

4.8. Renegotiation Attack

The Renegotiation Attack is an attack discovered by Marsh Ray and Steve Dispensa in 2009 [64]. The attack allows an attacker to perform a chosen prefix attack. To execute the attack, the attacker first establishes a connection with a server. Then, the attacker transmits arbitrary plaintext. The attacker then waits for an incoming connection of a victim. The attacker then encrypts the messages from the victim to the server as if they were coming from the attacker and decrypts the messages coming from the server and forwards them to the client. The attacker does this till the `ChangeCipherSpec` message was

transmitted in either direction; from that point forward the attacker only forwards the messages from either party. From the client's perspective, the client simply created a connection with the server, while from the server's point of view, the attacker simply renegotiated a connection with the attacker. However, what really happened is that the attacker forced the client to renegotiate the connection of the attacker. While the attacker does not have the keys for this new connection, the attacker could already send an arbitrary suffix before the client's application data reached the server. The whole attack, therefore, achieves something very similar to a session fixation attack.

4.9. Triple Handshake

The triple handshake attack [65] is one of the latest attacks on the TLS protocol and abuses multiple TLS features at the same time to re-enable a variation of the renegotiation attack, with the mitigation for the renegotiation attack in place. The triple handshake attack assumes an attacker model, where the attacker is a MitM and MitB and also an operator of a legitimate website with which the user would use client authentication to authenticate itself. The main goal of the attack is to trick the client into creating a connection with the attacker and to make the client believe that it is performing delayed client authentication with the attacker, while actually, it is authenticating itself with another server.

The attack starts with the client establishing a connection with the server of the attacker. The attacker then simultaneously starts its own connection with a victim server. The goal of the attacker is to synchronize the master secrets of the two connections, visualized in Figure 4.3. The attacker ensures that his connection to the victim server uses RSA as a key exchange algorithm and uses the same client random as the client. The attacker in return uses the same server random as the victim server in its `ServerHello` message to the client and uses an RSA cipher suite as well with the same session ID. The `ClientKeyExchange` the client then sends is decrypted by the server of the attacker, and the containing PMS is then used in the connection from the attacker to the victim server. This results in both connections having the same MS, as the client and server random as well as the PMS are the same. However, at this stage, the attacker did not gain anything, since the victim thinks that it connected to the attacker and the attacker just issued its own connection with the target server. This is where the second stage of the attack starts. Here the attacker terminates the connection and reconnects using session resumption. The client resumes the session with the attacker and the attacker resumes the session with the victim server. The attacker synchronizes the client and server random for both handshakes as before. This results in yet another connection where both MSs are equal, but in contrast to the first connection, this connection also has the `verify_data` of both connections synchronized. From here on, the attacker proceeds with the third stage of the attack, where the attacker can send arbitrary plaintext to the client (while the client still talks with the attacker) before the attacker initiates a renegotiation. This renegotiation now uses delayed client authentication. From this point on, the attacker acts passive and simply

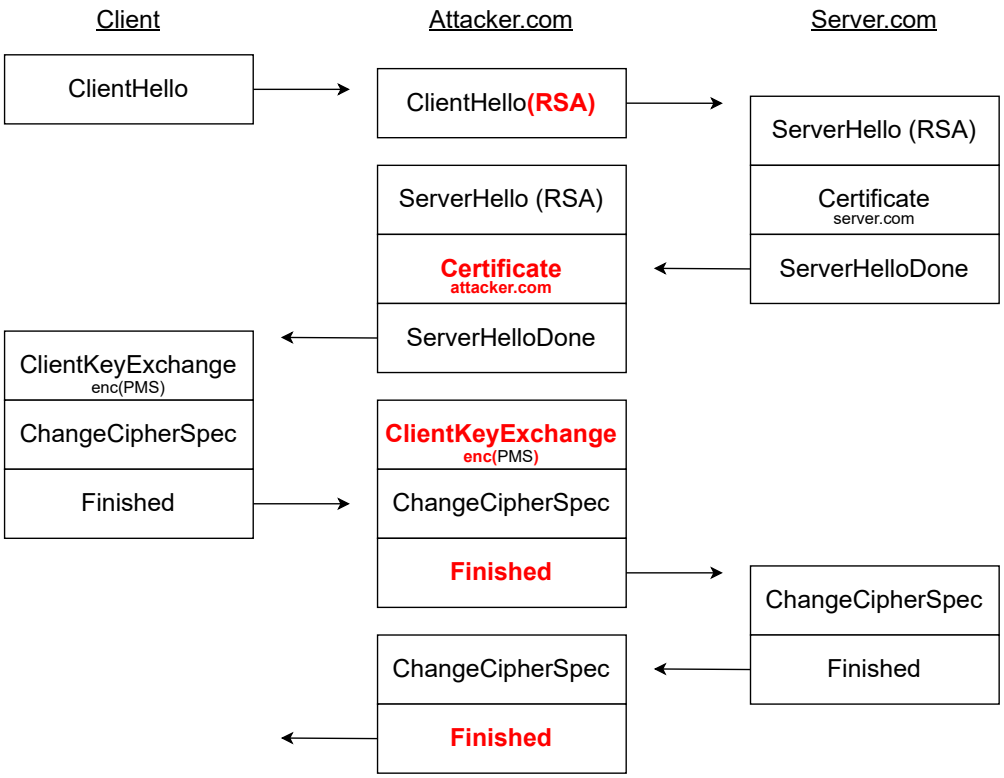


Figure 4.3.: Master secret synchronization in the Triple Handshake attack. Modifications done to the messages are highlighted in red, otherwise the attacker forwards the same messages the client sends in its own connection to a server. Most noteworthy, the attacker sends the same PMS in its `ClientKeyExchange` message, resulting in the same master secret.

forwards the messages and re-encrypts them if necessary until the handshake is completed. Note that due to the delayed client authentication, the client thinks it just authenticated at the attacker's server but actually did at the victim's server. This is possible, since typically in renegotiation handshakes, clients do not verify the identity in the server certificate anymore. At this point, similar to the renegotiation attack, the attacker does not have the keys to the final session anymore. The attack is not prevented by the renegotiation extension, as the attacker synchronized the `verify_data` by resuming the session first.

The attack was mitigated by the introduction of the *extended master secret* extension, which lets the whole transcript influence the MS computation, and therefore makes it impossible for an attacker to synchronize the master secret of the connection.

While the attack relies on the client not validating the server's identity in the renegotiation handshake (again), the attack is still very impressive, as it combines almost all TLS workflows within a single attack.

4.10. SLOTH

The SLOTH attack [97] is a series of attacks that explore the impact of hash collision attacks in TLS. Concretely, the attack showed that the use of MD5 signature algorithms could drastically weaken the security of client and server authentication or when *tls-unique* [98] is used. The attack resulted in the deprecation of MD5-based signature algorithms.

4.11. BERserk

The BERserk attack [35] is a variant of the Bleichenbacher06 attack [34] on small RSA exponents (e=3) that exploited a vulnerability in NSS. The ASN.1 parsing of NSS was too lenient and allowed an attacker to create signatures that were treated as valid by NSS. After the attack, certificates with small RSA exponents were mostly removed from the Internet PKI to prevent similar attacks in the future.

4.12. Invalid Curve Attacks

A common flaw in elliptic curve implementations can lead to invalid curve attacks [99]. If an implementation wants to perform any operation involving secret key material on an elliptic curve point, it should verify that the point is actually on the elliptic curve. If it does not, it risks that it performs a computation on a point that is not on the curve, and therefore might leak information about secret key material. Concretely, an attacker may send a group element that forms a small subgroup, for which it is trivial to find a solution (a congruence). Given enough congruences, the attacker can compute the private key. To retrieve enough information to compute the private key, it is important that the attacker can force the target to redo the computation multiple times under the same private key. The attack was analyzed in the context of TLS by Jager et al [100].

4.13. FREAK

The FREAK attack [101] is a (then) common implementation flaw in the state machine of TLS implementations. The flaw treated incoming server key exchange messages in a non-export cipher suite, as if an export cipher suite was negotiated. This would allow an attacker to negotiate weak parameters with a vulnerable client and a server that supports RSA-export cipher suites. The attack ultimately resulted in the removal of RSA-export cipher suites from server configurations.

5

Raccoon Attack

In this chapter, the Raccoon attack is introduced. The Raccoon attack is a novel timing side-channel attack based on a flaw in the TLS specification that I discovered during my work on the protocol. The Raccoon attack was published at USENIX'21 [9] and is a joint research project between the Ruhr University Bochum, the School of Computer Science at Tel Aviv University, Paderborn University and the Bundesamt für Sicherheit in der Informationstechnik (BSI). The HNP computations and experiments from Section 5.14 were done by Marcus Brinkmann and Johannes Mittmann. Subsection 5.13.1 was mostly done by Nimrod Aviram, where I provided the measurements for the full Raccoon connections. Though my own contributions to these specific subsections are not significant, I included them to improve the readability and completeness of this dissertation. The project was further assisted by Juraj Somorovsky and Jörg Schwenk.

5.1. Details on Hash Functions

To understand the Raccoon attack in full detail, we first need to establish additional background information about the internal processing or blocks on the hash functions used in the PRFs of SSL/TLS Table 5.1 gives an overview of hash functions. The second and third columns indicate the input and output block size, respectively. The fourth column provides the minimum number of bytes appended that are appended internally to the input. For example, when using SHA256 (which uses a block size of 64 bytes), at least 9 bytes have to be appended to the input message. Therefore, messages of up to 55 bytes will be processed as one block, using two calls to the compression function (due to the finalization function). Messages of length between 56 and $128 - 9 = 119$ bytes will be processed as two input blocks, using three calls to the compression function. Table 5.1 provides further examples for input block boundaries.

Hashfunction	Block size	Output	Len. and pad.	Input blockborders
MD5	64	16	8+1	55, 119, 183, ...
SHA1	64	20	8+1	55, 119, 183, ...
SHA256	64	32	8+1	55, 119, 183, ...
SHA384	128	48	16+1	111, 239, 367, ...

Table 5.1.: Properties of common hash functions. All values are denoted in bytes.

5.2. Raccoon Length Distinguishing Oracles

To explain the Raccoon attack we will first look at length distinguishing side channel oracles in TLS-DH(E). On their own, these side channel oracles can be used to perform length distinguishing attacks. As we will later show, these length distinguishing attacks can be used to perform a PMS recovery attack.

The length distinguishing side channel oracles exploit the following fact:

> *The key derivation function KDF strips leading zeros from the computed DH secret g^{ab} and performs further computations based on the modified secret string.*

These computations can result in different timing behaviors based on the number of removed bits or different error behavior. An attacker observing the timing behavior can construct an oracle from the behavior of an application using Diffie-Hellman (DH) key exchange and use it to leak some of the most significant bits (MSB) of the shared secret.

This already invalidates the standard indistinguishability assumption of the cryptographic primitives used (DDH [102], PRF-ODH [103]).

We define $\mathcal{O}_{k,b}(x)$ as an oracle that reveals if the k most significant bits of the n-bit number $x^b \bmod p$ are zero:

$$\mathcal{O}_{k,b}(x) = \begin{cases} \text{True} & \text{if } \mathrm{MSB}_k(x^b \bmod p) = 0\,, \\ \text{False} & \text{otherwise}\,. \end{cases} \tag{5.1}$$

The effective number of bits leaked from this oracle depends on the modulus length and bias, the underlying KDF properties, as well as implementation details, and can range from a fraction of a bit to several bytes in case the result is True. In the following subsections, we give four different constructions $\mathcal{O}^{\mathrm{H}}$, $\mathcal{O}^{\mathrm{C}}$, $\mathcal{O}^{\mathrm{P}}$, and $\mathcal{O}^{\mathrm{D}}$for such oracles. Then in Section 5.8, we instantiate such oracles in the context of TLS servers and show in Section 5.14 how they can be used to run a full attack to uncover the complete PMS.

5.3. $\mathcal{O}^{\mathrm{H}}$ Hash Function Invocation

In HMAC constructions (RFC 2104 [104]), the shared secret key g^{ab} may either be used directly in the HMAC computation (if $|g^{ab}|$ is smaller than the maximal HMAC key size), or it must be hashed to a smaller size.

Consider a server that uses a DH prime modulus p with $|p| = 1025$ bits and a PRF based on HMAC-SHA384.

For this PRF, the secret key k can at most be 128 bytes long, which is the input block size of the hash function SHA384 (Table 5.1). For this purpose, the KDF first strips leading zero bits and then converts K to a byte sequence.

Now the KDF program branches:

1. If the length of the byte sequence is at most 128 bytes, this byte sequence is used directly as the HMAC key k.

2. If the length of the byte sequence is bigger than 128 bytes, *the SHA384 hash function is invoked once on this byte sequence*, and the resulting hash value, padded with 80 zero bytes, is used as the HMAC key: $k = h||0x0...0$.

Now assume that a MitM attacker observed a DH key exchange. The goal of the attacker is to learn the first bit of $K = g^{ab} \bmod p$.

As described above, there are two possibilities for a server-side KDF to process the shared secret $K = g^{ab}$:

- The most significant bit of K is 0. The server *strips the leading zero bit* and converts K to a byte array which will consist of 128 bytes. Since the byte array is 128 bytes long, it is directly used in the HMAC computation: $\mathrm{HMAC}_{\mathrm{SHA384}}(K, \mathit{seed})$.

- The most significant bit of K is 1. The server converts K to a byte array consisting of 129 bytes. A 129-byte long shared secret cannot be directly used in the HMAC computation (see also Subsection 2.7.1); before computing HMAC, the server needs to compute SHA384 over K. It can then use the SHA384 output as an input for the HMAC computation.

Observe how a shared secret K starting with 1 results in an additional SHA384 *hash function invocation* over K.

In the previous example, the modulus was exactly one bit bigger than the block size of the hash function and leaked only the most significant bit of the PMS. If the modulus is k bits bigger than the block size, the attacker has a chance of $1/2^{\ell}$ to leak the top k bits of the PMS. As we show in Section 5.13, this timing difference is observable by a remote attacker.

5.4. $\mathcal{O}^{\mathrm{C}}$ Compression Function Invocations

This oracle exploits the number of invocations of the internal compression function if the second branch in $\mathcal{O}^{\mathrm{H}}$ occurs, i.e., if the shared DH secret $K = g^{ab}$ is bigger than the input block length of the HMAC hash function.

As mentioned in Section 2.6, hash functions based on the Merkle-Damgård scheme operate on blocks. The number of blocks a hash function has to process depends on the input length (see Table 5.1). If the DH shared secret K is used as a key for an HMAC computation, it can have distinct timing profiles depending on its length.

To give an example for HMAC-SHA384, consider a 1913-bit DH modulus p encoded in 240 bytes. The server-side KDF implementation now has to invoke the hash function over the shared key K, since K is much larger than the allowed 128 bytes. We now get an MSB oracle from the number of compression function invocations:

1. If the most significant bit of K is 0, K will be coded into 239 bytes. With the 17 bytes added for length and padding (cf. Table 5.1), it will fit into two blocks. Thus, the server will execute three hash compressions.

2. If the most significant bit of K is 1, K will be decoded into 240 bytes. Appending padding and the length field will fit into three blocks; the server will execute four hash compressions.

Analogously to the previous oracle, if the modulus is k bits bigger than a critical block border, the attacker has a chance of $1/2^{\ell}$ to leak the top k bits of the PMS, where $\ell = k - \epsilon$ (see Section 2.14).

5.5. $\mathcal{O}^{\mathrm{P}}$ Key Padding

Another side channel arises based on the number of padding bytes used to pad the DH shared key. The HMAC interface [104] pads keys to the block size of the hash function. The padding of the shared key can result in a timing side channel as different key lengths will lead to different amounts of padding applied and therefore to a different number of calls to the hash compression function. We show a practical attack based on this side channel in Subsection 5.13.2.

5.6. $\mathcal{O}^{\mathrm{D}}$ Direct Side Channels

Until now, we discussed side channels based on small timing differences in the processing of the shared DH secret. However, it is possible that an implementation provides a direct oracle that does not rely on timing differences but relies on direct differences in behavior, such as error messages or handling of the connection state (like closing the underlying socket). If an implementation behaves differently depending on the shared secret, it provides an attacker with a direct side channel. These direct oracles are plausible because the zero byte is considered a special character in many programming languages. For example, in C the zero byte is used to terminate strings. This can result in programming errors, which can, in return, lead to observable differences in response to network queries.

We show in Section 9.1 that a non-negligible number of real-world servers indeed present such directly observable behavior differences. In all observed cases, this side channel only leaked the most significant byte of the PMS, which is equivalent to a leak of $k = n \bmod 8$ bits for a prime p of bit-size n.

5.7. Further Oracle Considerations

Big Number Libraries. Even if a protocol does maintain leading zero bytes of the shared secret, the used big number library might introduce an oracle that leaks the most significant bits. If the big number library does not maintain fixed-size big numbers internally, the resulting shared secret has to be padded by the application to the modulus size if the shared secret has fewer bytes than the modulus.

An example of this side channel in OpenSSL can be found in Subsection 5.13.3.

Hitting the Block Boundaries with Dangerous Modulus Sizes. In our previous examples, we used unusual modulus sizes of 1025 and 1913 bits to instantiate the length distinguishing oracles. We arrived at these numbers by computing the input lengths for a given hash function that leaks the top x leading zero bits of the potential input at the *critical block border* of the nth block, using the following formula: $cbb(x, b, p, n) = n * b - p + x$, where b is the block size of the hash function in bits, and p is the fixed padding part of the hash function, also in bits.

On the Reliability of Timing Side Channels. If the attacker uses a timing side channel, the oracle will likely occasionally give wrong results, as timing measurements are inherently noisy. Thus, any classifier will exhibit some probability of false classification. The distinguishing attack can be made practical if the attacker can send several queries to the target. The attacker can then use standard statistical tests to build a reliable oracle out of the noisy oracle. More details on this are provided in Subsection 5.13.1.

5.8. TLS Attack Scenarios

For the attack scenarios described below, the attacker needs access to a functional oracle from Chapter 5. Furthermore, the honest client and server have to use a vulnerable TLS version and negotiate TLS-DHE or a connection with a static TLS key share.

Raccoon^e: Length Distinguishing Attack on Ephemeral Keys. The goal of the Raccoon^e attack is to detect the leading bits in the DH shared secret in a MitM attacker model with ephemeral keys. If the attacker wants to perform the attack, it can measure the presented side channels in Chapter 5 at two different positions within a TLS connection:

- The attacker can target the server and measure the time the server used to compute the PMS. The attacker can do this by measuring the time between the server receiving the `ClientKeyExchange` message and the server sending its `Finished` message.
- Or, the attacker can target the client and measure the time the client used to compute the PMS. The attacker can do this by measuring the time the

client took to read the `ServerKeyExchange` message up to sending the `Finished` message by the client.

By repeatedly observing TLS-DHE handshakes between an honest client A and an honest server B the attacker can learn typical timing values. After this, the attacker will be able to detect if leading zero bytes are present in the unknown PMS by observing faster response times.

This length distinguishing attack is applicable even if the server does not reuse ephemeral DH values. However, in this case, the attack poses little threat in practice, since the attacker merely learns the length of a fully ephemeral, one-time shared secret. This does not allow the attacker to decrypt or modify traffic.

Raccoon^s: Length Distinguishing Attack on a Static Key. In this scenario, the attacker has recorded a previous TLS-DH(E) session, and the goal is to recover the length of the PMS used in this session between two honest peers. In contrast to Raccoon^e, in this scenario, the server uses a static key or is reusing the same ephemeral DH secret for a certain period of time, covering the recorded TLS-DHE session and the full duration of the attack.

To perform the attack, the attacker selects an appropriate oracle of $\mathcal{O}^{\mathrm{H}}$, $\mathcal{O}^{\mathrm{C}}$, $\mathcal{O}^{\mathrm{P}}$, or $\mathcal{O}^{\mathrm{D}}$ from Chapter 5, connects to the server, and sends a Diffie-Hellman share in a `ClientKeyExchange` message. For the length distinguishing attack, this `ClientKeyExchange` message contains the originally observed key share from the honest client. Note that an attacker can also send related key shares here to retrieve the MSBs of related PMSs (see Section 5.14). Of course, the attacker cannot construct a valid `Finished` message since the secret key is unknown to the attacker. The server receiving a message crafted by the attacker will terminate the connection by either sending a fatal `Alert` message or closing the TCP connection. However, the server always needs to compute the PMS and derive the MS using the PRF. Therefore, the server's response will depend on the leading bits of the PMS.

If the attacker uses a timing side channel, the reliability of the side channel can be improved as described in Section 5.7.

5.9. Analysis of TLS Key Derivations

Since the TLS key derivation is of special interest for the attack, we will analyze it in detail. The starting point for the key derivation is the PMS. In a two-step key derivation, first, a *MS* is computed from this PMS, and then two sets of keys (one for each communication direction) are derived from the MS.

How exactly the MS is derived from the PMS depends on the negotiated protocol version and cipher suite. Note that an attacker can observe the `ClientKeyExchange` message on any version or cipher suite and then send it as part of a different protocol version and cipher suite to a server (as long as the server supports it). We now analyze different TLS versions and how they use the PMS to derive further keys with their PRF's. Our analysis of critical block borders is summarized in Table 5.2.

Protocol version	Key derivation	Critical *PMS* comp. block borders
TLS 1.2 (SHA384)	SHA384 PRF	128, 239, 367, ...
TLS 1.2 (others)	SHA256 PRF	64, 119, 183, ...
TLS 1.0 and 1.1	MD5/SHA1 PRF	110, 238, 366, ...
SSLv3	Custom MD5/SHA1	45, 54, 55, 56, 99, 118, 119, 120, ...

Table 5.2.: Key derivation properties of non-PSK cipher suites. The first and second columns provide the protocol version, cipher suite, and the hash algorithms used in the key derivation function. The last column provides critical block borders for a PMS in bytes. For example, a 239-byte long PMS consumes one less SHA384 hash compression than a 240 bytes long pms.

TLS 1.2. In TLS 1.2 the MS is derived from an HMAC-based PRF construction. The MS is computed as:

$$ms = \mathrm{PRF}(pms, label, ClientRandom \,||\, ServerRandom)\,.$$

The PMS will be used as a key for HMAC operations within the PRF. The used HMAC depends on the selected cipher suite. Per default, SHA256 is used, but the cipher suite could also specify the usage of SHA384 (if the cipher suite name ends with `_SHA384`). For TLS 1.2 the side channel analysis of Chapter 5 can be directly applied. The PMS maximum size is the size of the DH key. In configurations with recommended DH key sizes larger than 2000 bits, the computed PMS will with overwhelming probability be larger than the block border (64 bytes for SHA256 and 128 bytes for SHA384). If the PMS is larger than the block size of the hash function, it must be hashed before using it in the HMAC computation. This potentially enables a side channel based on the number of hash compression function invocations (cf. Section 5.4).

Note that in the case of SHA384-PRFs with DH key sizes slightly bigger than 1024 bits, the hash function invocation side channel and the resulting oracle $\mathcal{O}^{\mathrm{H}}$ can be used (see Section 5.3).

TLS 1.0 and TLS 1.1. These two protocol versions use the same PRF, which is based on a combination of SHA1 and MD5. In this PRF, the PMS is split into two halves: The first half enters an expansion function based on MD5, while the second half enters a distinct key expansion function based on SHA1. The final output of the TLS 1.0 and 1.1 PRF is the XOR of these two expansion functions. If the PMS has an odd number of bytes, the byte in the middle of the PMS will be used by both halves.

Since TLS 1.0 and TLS 1.1 split the shared secret into two halves, the computations for inputs that reach the block borders change in comparison to TLS 1.2, as each hash function adds its own padding and length bytes internally. Note that since two hash functions are used at the same time (with identical input lengths and hash function properties such as input block size, length, and padding, see Table 5.1), the created side channel is amplified. For TLS 1.0 and TLS 1.1 the size of inputs that leak the top x leading zero bytes at the nth

block border can be computed with the formula

$$cbb_{\text{TLS1.0/1.1}}(x, n) = (64n - 9) \cdot 2 + x\,, \tag{5.2}$$

where x is the number of most significant bytes to be leaked.

SSLv3. Even though SSLv3 is deprecated, some servers on the web still support it.[1] SSLv3 key derivation is strictly different from the key derivation used in TLS. While the leading zero bytes from the PMS are stripped, the MS is then computed as

$$\begin{aligned} ms :=& \text{MD5}(pms \,||\, \text{SHA1}(pms \,||\, \text{"}A\text{"} \,||\, r1 \,||\, r2)) \,||\\ & \text{MD5}(pms \,||\, \text{SHA1}(pms \,||\, \text{"}BB\text{"} \,||\, r1 \,||\, r2)) \,||\\ & \text{MD5}(pms \,||\, \text{SHA1}(pms \,||\, \text{"}CCC\text{"} \,||\, r1 \,||\, r2))\,, \end{aligned} \tag{5.3}$$

where $r1 := \mathit{ClientRandom}$ and $r2 := \mathit{ServerRandom}$.

This computation results in more opportunities for an attacker to construct a possible side channel from an additional hash function compression invocation. The outer MD5 functions hash the shared secret in concatenation with the output of the inner SHA1 function. The outer function adds an offset of 20 bytes to the shared secret. As this operation is done three times, the side channel within the MD5 computation is amplified by a factor of three. The inner SHA1 computation hashes different inputs each time. The first call hashes a label of length 1, while the second call hashes a label of length 2, and the last call hashes a label of length 3. Each time two (32-byte long) random values of the client and server are hashed as well. This generates a total offset of 65, 66, and 67 bytes, respectively. The resulting inputs (in bytes) which leak the top x leading zero bytes at the nth block in SSLv3 can therefore be computed as:

$$\begin{aligned} cbb_{SSL}(x, n) &= 64n - (9 + 20) + x \\ cbb_{SSL-A}(x, n) &= 64n - (9 + 65) + x\,; \quad n > 1 \\ cbb_{SSL-BB}(x, n) &= 64n - (9 + 66) + x\,; \quad n > 1 \\ cbb_{SSL-CCC}(x, n) &= 64n - (9 + 67) + x\,; \quad n > 1 \end{aligned} \tag{5.4}$$

TLS DHE-PSK. Although not as widespread, TLS also offers a variety of cipher suites that allow the usage of preshared keys (PSK) [105]. In DHE-PSK, the client basically performs the same handshake as a normal DHE handshake, resulting in the shared DH value $g^{ab} \bmod p$. Then, both client and server authenticate using a PMS which is computed based on the preshared key PSK as

$$pms := \text{len}(g^{ab} \bmod p) \,||\, g^{ab} \bmod p \,||\, \text{len}(\text{PSK}) \,||\, \text{PSK}\,, \tag{5.5}$$

where $\text{len}(x)$ indicates a two-byte length value of x (in bytes).

Since DHE-PSK changes the way the PMS is computed, the block borders for the Raccoon attack change as well. Interestingly, the block borders depend on

[1] According to the SSL pulse measurements of September 2020, SSLv3 is supported by 4.4% of the servers from the Alexa top 150k list.

the length of the preshared key PSK. DHE-PSK shifts the length of the PMS, which enters the PRF by $4 + |PSK|$ bytes. This can bring otherwise unfeasible modulus sizes in proximity to the critical block border for the attacker. An attacker being able to set a PSK for an arbitrary, attacker-controlled identity could therefore choose a PSK to reach the advantageous critical block boundaries. A related side channel exists in SSH and is described in Section 5.15.

If the attacker is not an authenticated user, they could use the DHE-PSK PMS processing within the PRF to perform a different length-distinguishing attack. Since the PSK length also directly influences the PRF computation time, the server response time could be used to determine the length of the PSK. Note that this is possible even if the server does not repeat the DH public keys and strictly uses ephemeral keys in DHE-PSK.

5.10. Dangerous TLS Modulus Sizes

Since the server chooses the modulus size and the attacker has no variable-size inputs to the PRF (except for DHE-PSK cipher suites), the attacker cannot influence the block borders of the hash function and thus optimize their usability as a side channel. Usually servers choose moduli whose lengths are of the form 2^n, like $2^{10} = 1024$, $2^{11} = 2048$ or $2^{12} = 4096$. The server is free to deviate from these and move to arbitrary sizes. For common bit lengths 2^n, the block border will never realistically reach a critical block border as the PMS would require too many leading zero bits. However, if a server deviates from these common modulus sizes, it can become possible for an attacker to hit the critical block borders. For example, LibTomCrypt[2] used to create 1036 bit moduli, which would make $\mathcal{O}^{\mathrm{H}}$ feasible for $k = 12$.

A lists of dangerous modulus sizes can be found in Table 5.3.

[2]https://github.com/libtom/libtomcrypt

TLS 1.2 (SHA384)	TLS 1.2 (other)	TLS 1.0/1.1 (MD5/SHA1)	SSLv3 (MD5)	SSLv3 (A)	SSLv3 (BB)	SSLv3 (CCC)
1025 - 1056	513 - 544	1025 - 1056	281 - 312	433 - 464	425 - 456	417 - 448
1913 - 1944	953 - 984	1905 - 1936	793 - 824	945 - 976	937 - 968	929 - 960
2937 - 2968	1465 - 1496	2929 - 2960	1305 - 1336	1457 - 1488	1449 - 1480	1441 - 1472
3961 - 3992	1977 - 2008	3953 - 3984	1817 - 1848	1969 - 2000	1961 - 1992	1953 - 1984
4985 - 5016	2489 - 2520	4977 - 5008	2329 - 2360	2481 - 2512	2473 - 2504	2465 - 2496
6009 - 6040	3001 - 3032	6001 - 6032	2841 - 2872	2993 - 3024	2985 - 3016	2977 - 3008
7033 - 7064	3513 - 3544	7025 - 7056	3353 - 3384	3505 - 3536	3497 - 3528	3489 - 3520
8057 - 8088	4025 - 4056	8049 - 8080	3865 - 3896	4017 - 4048	4009 - 4040	4001 - 4032
...	4537 - 4568	...	4377 - 4408	4529 - 4560	4521 - 4552	4513 - 4544
	5049 - 5080		4889 - 4920	5041 - 5072	5033 - 5064	5025 - 5056
	5561 - 5592		5401 - 5432	5553 - 5584	5545 - 5576	5537 - 5568
	6073 - 6104		5913 - 5944	6065 - 6096	6057 - 6088	6049 - 6080
	6585 - 6616		6425 - 6456	6577 - 6608	6569 - 6600	6561 - 6592
	7097 - 7128		6937 - 6968	7089 - 7120	7081 - 7112	7073 - 7104
	7609 - 7640		7449 - 7480	7601 - 7632	7593 - 7624	7585 - 7616
	8121 - 8152		7961 - 7992	8113 - 8144	8105 - 8136	8097 - 8128
	...		...	...	...	...

Table 5.3.: A list of dangerous TLS modulus sizes (in bits). A modulus has to be considered as dangerous if the mentioned parameter combination is supported by the server. It is not important which parameter combination was negotiated by the attacked connection. The list considers an attacker who wants to leak 1-32 bits at a block border.

5.11. Raccoon Premaster Secret Recovery Attack

Until now, we have discussed a distinguishing attack on TLS, which allows an attacker to detect leading zero bytes of the PMS. If the server reuses the DH values for multiple connections (cf. [106]), the distinguishing attack can be turned into a full PMS recovery attack. This is the case for TLS-DH and for TLS-DHE if the server disregards best practices and reuses ephemeral keys. Our attack is based on the well-known Hidden Number Problem described by Boneh and Venkatesan [13] (see Section 2.14).

We use the attack scenario **Raccoon**^{s} from Section 5.8 which leaks the top k bits of the PMS. We assume the server reuses the same secret DH exponent b; this reuse does usually not depend on the TLS version or cipher suite (except for export cipher suites not considered here), so our **Raccoon**^{s} attacker can choose a beneficial TLS version and cipher suite for the attack, as long as they are supported by the server. The attack proceeds in four phases:

Phase 1: Passive MitM. In this phase, the attacker records a complete TLS-DH(E) session and extracts g^a from the `ClientKeyExchange` message as well as g^b, g, and p from the `ServerKeyExchange` or `Certificate` message.

Phase 2: Active Web Attacker. In this phase, the attacker interacts as a client with the server. However, instead of choosing a secret ephemeral DH value a' and sending $g^{a'}$ in the `ClientKeyExchange` message, the attacker chooses random values $r_i \in \mathbb{Z}_q$ and sends the value $x_i = g^a g^{r_i}$ in the `ClientKeyExchange` message (cf. Figure 5.1). To finish this part of the TLS handshake,

the attacker sends a `ChangeCipherSpec` and the client's `Finished` message, where the content of the `Finished` message is chosen randomly because the attacker lacks the keys (MS and the symmetric keys) to compute a valid `Finished` message correctly.

After sending `ClientKeyExchange`, the attacker starts measuring a chosen length distinguishing oracle $\mathcal{O}^{\mathrm{H}}$, $\mathcal{O}^{\mathrm{C}}$, $\mathcal{O}^{\mathrm{P}}$, or $\mathcal{O}^{\mathrm{D}}$, until some `Alert` message arrives from the server. The attacker may repeat the measurement by using the same value r_i for multiple measurements until some statistical test (e.g., Mann-Whitney [107]) indicates that a sufficiently high probability level has been reached. If the measurement indicates that the leading k bits have been stripped, we have found a candidate r_i for an HNP equation.

If the measured time indicates that less than k bits have been stripped, a new random value r_{i+1} is chosen by the attacker and phase 2 is repeated.

We describe another attack scenario against the unused static-DH client authentication in Section 5.12.

Phase 3: Constructing an Instance of HNP. Now that the attacker has learned that for the candidate r_i the oracle

$\mathcal{O}_{k,b}(r_i)$ is True, the attacker knows that $0 < {r_i}^b \bmod p = g^{ab} g^{br_i} \bmod p < 2^{n-k}$. Subtracting 2^{n-k-1}, we obtain the centered equation

$$\left|\alpha \cdot t_i \bmod p - y_i\right| < 2^{n-k-1} = p/2^{\ell+1}\,, \tag{5.6}$$

where $\alpha := g^{ab} \bmod p$ is unknown (the *hidden number*) and $t_i := (g^b)^{r_i} \bmod p$, $y_i := 2^{n-k-1}$ are known to the attacker.

Equation 5.6 corresponds to the randomized version of HNP as defined by Boneh and Venkatesan [13], except that in our case the oracle does not reveal the MSBs directly, but only whether they are zero or not. Moreover, we center the equation around zero and take the bias of p into account, as in Nguyen et al [47].

Phase 4: Computing the Premaster Secret. Phases 2 and 3 are repeated until the attacker has obtained a sufficient number of equations to solve the HNP instance and recovers the hidden number g^{ab}, which is the PMS of the connection the attacker observed in phase 1 of the attack. We will show in Section 5.14 that this is indeed possible. With the PMS the attacker can then derive the MS; with the MS the attacker can proceed to compute the symmetric keys and decrypt the connection.

5.12. Attacking Static-DH Client Authentication

Although not used in practice, TLS also offers the possibility of client authentication with static-DH certificates. If the server requests client authentication and the client has a static-DH client certificate that matches the parameters the server has chosen, the client will not send a fresh public key in the `ClientKeyExchange` message. Instead, the long-term public key from the certificate will be used for the PMS generation. This enables a variation of the presented

attack that could be executed by an attacker if the client is using the same certificate with an attacker-controlled server. In this attack scenario, the attacker observes a handshake with static-DH client authentication he wants to decrypt. In the second stage, he forces the client to repeatedly connect to his own server and negotiate static-DH with client authentication. Similar to the original Raccoon attack, the attacker now chooses this server's static-DH public key to be $g^b \cdot g^r \bmod p$. This time, the client computes $g^{ba} \cdot g^{ra} \bmod p$ and sends his `Finished` message. The attacker measures the time the client took to compute the PMS. He can then use one of the MSB oracles to retrieve the MSB of the shared secret and create and solve an instance of the HNP.

5.13. Evaluation

In this subsection, we will analyze if the requirements of the Raccoon attack can actually be fulfilled by a real-world attacker, namely, measuring the timing difference by the created side channel and solving the HNP for real modulus sizes with realistic leak sizes.

5.13.1. Timing Measurements

As demonstrated by the Lucky 13 attack [11], applying a hash function to inputs of varying lengths results in a measurable difference in processing times. We now shortly revisit this finding by evaluating the OpenSSL library (version 1.1.1), before putting it in the context of our attack.

Figure 5.1 shows a plot of the processing time (in cycles) to compute HMAC with SHA256 and SHA384 for keys of varying lengths, on 1024-byte messages. To simplify the presentation, we report the median processing time across 10,000 experiments per input length. The step-like increase in processing time as the key size increases can clearly be seen. The first step in the increase of processing time leads to oracle $\mathcal{O}^{\mathrm{H}}$, the hash function invocation oracle. The subsequent, slightly smaller steps lead to oracle $\mathcal{O}^{\mathrm{C}}$, the compression function invocation oracle. The smallest visible steps (for SHA256, when the input length is $128 \cdot k - i, 1 \leq i \leq 8$, and similarly for SHA384) lead to oracle $\mathcal{O}^{\mathrm{P}}$; we analyze the cost of exploiting this side channel in Subsection 5.13.2.

Is the difference in processing times measurable in a remote setting? To measure if the side channel is big enough for a remote attacker, we created a test setup consisting of two (non-virtual) machines, one simulating the attacker machine and one simulating a victim server. The machines are directly connected with a 1 Gbit/s connection. The attacker machine used an Exablaze ExaNIC HPT network adapter. This network card is specifically built to generate high-precision hardware timestamps.

For the evaluation, a tool on the attacker machine repeatedly performed handshakes with the victim TLS server. The tool generated a DH private key and computed the resulting DH shared secret, alternating between handshakes where the DH secret starts with a single leading zero byte or no leading zero bytes. For

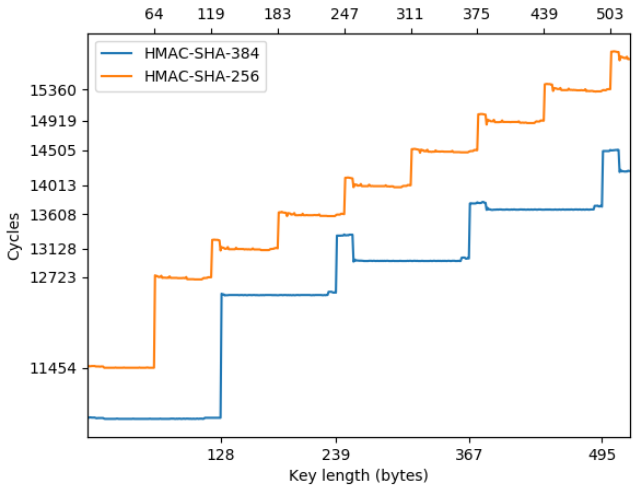


Figure 5.1.: Processing time to compute HMAC-SHA256 and HMAC-SHA384 with keys of varying lengths for inputs 1KB in length, measured in CPU cycles. Reported values are medians across 10,000 experiments per key length, performed with OpenSSL version 1.1.1.

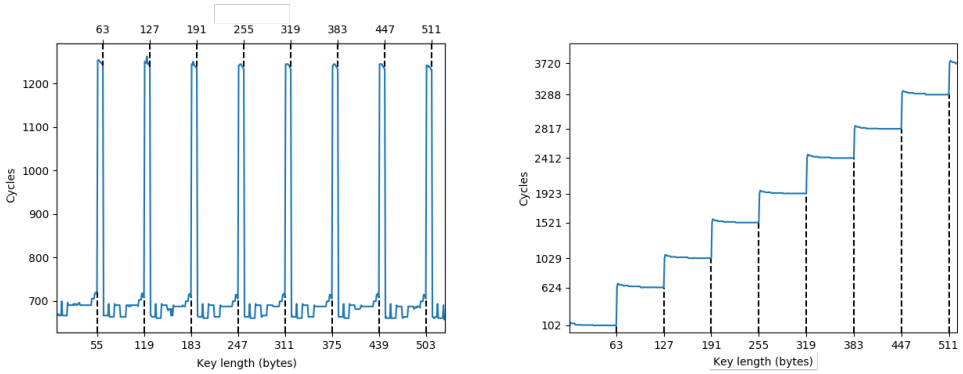
each handshake, the tool recorded if the MSBs of the DH secret were zero, as well as the server's response time. We used a modulus size of 1032 bits to analyze whether the side channel is measurable, as this creates the hash function invocation side channel $\mathcal{O}^{\mathrm{H}}$ (see Section 5.3). We collected 100,000 measurements each for PMSs with a leading zero byte and without a leading zero byte.

In broad terms, the attacker would use a classifier to approximate the oracle's response. That is, the attacker collects server response times from handshakes using DH share g^{a+r}, and attempts to deduce from these measurements the oracle response $\mathcal{O}^{\mathrm{H}}(g^{a+r})$. Any classifier will exhibit some probability of false classification. False negatives occur when the classifier concludes that $\mathcal{O}^{\mathrm{H}}(x) =$ False when $\mathcal{O}^{\mathrm{H}}(x) =$ True. Similarly, false positives occur when the classifier wrongly concludes $\mathcal{O}^{\mathrm{H}}(x) =$ True when $\mathcal{O}^{\mathrm{H}}(x) =$ False.

In our experiments, the Mann-Whitney test [107] performed very well for distinguishing between the two cases. This test can be configured with a desired false positive probability, which then determines the (empirical) false negative probability. With 100 samples per case (200 measurements overall) and a 10% false positive rate, the false negative rate was 10.4% (we also empirically confirmed that the false positive rate is 10%). To estimate these false-reporting rates, we conducted 200,000 experiments, wherein In each experiment, the samples for each set were randomly selected from the pool of 100,000 collected samples. Increasing the number of samples to 1,000 (2,000 measurements overall) allowed us to achieve a false positive rate of 0.009200% and a false negative rate of 0.000795%.[3]

An attacker would have to account for the false reporting rates when perform-

[3]To estimate these lower rates, we ran our classifier on 20 million sets of randomly-sampled 2,000 measurements.



(a) Finalize (b) Update

Figure 5.2.: Running time of the SHA256 finalize and update function for inputs of varying lengths, measured in CPU cycles. Reported values are medians across 10,000 experiments per key length, performed with OpenSSL version 1.1.1.

.

ing the attack. To deal with false positives, the attacker re-measures timings for any reported positive. That is, the attacker first performs 200 measurements for each x value. For values where the classifier outputs $\mathcal{O}^{\mathrm{H}}(x) = \text{True}$, the attacker re-measures the processing time for x, obtaining 2,000 more measurements, and re-runs the classifier.

Iterating over a total of m DH values, in expectation at most $m \cdot 255/256$ values are true negatives,[4] of which $m \cdot 255/256 \cdot 10\% \cdot 0.009200\% = m \cdot 9.1 \cdot 10^{-6}$ will be falsely labeled as positives in both classification rounds. Similarly, $m/256$ are true positives, of which $m/256 \cdot (1 - 10.4\%) \cdot (1 - 0.000795\%) = m \cdot 0.35\%$ will be correctly labeled as positives in both classification rounds.

The attacker needs to collect roughly 180 true positive values to solve the HNP problem for a 1024-bit modulus (see Section 5.14). Choosing $m = 55,000$ results in 192 correctly identified positives in expectation, and 0.5 false positives. The overall required number of timing samples is therefore 22.34 million. These numbers are not necessarily optimal.

5.13.2. Exploiting $\mathcal{O}^{\mathrm{P}}$

Following Subsection 5.13.1 (see Figure 5.1), we now discuss mounting the attack using $\mathcal{O}^{\mathrm{P}}$, the key padding oracle. To recap: The computation time of HMAC-SHA256 exhibits small 'steps' when the input key length is $128 \cdot k - i, 1 \leq i \leq 8$. In particular, when using a 1024-bit (128 bytes) DH modulus, there is a measurable timing difference when the MSB is zero. Unlike previous oracles discussed in this work, this oracle allows attacking commonly-used modulus sizes, albeit at a much greater cost due to the smaller timing difference.

Our closer analysis revealed that this timing difference stems from the 'finalize'

[4] If we denote the most significant byte of the modulus as v, then $^{v-1}/_{256}$ shared secrets are true negatives. It is common for v to be smaller than 256, slightly lowering the attack cost, but we prefer to give a worst-case analysis.

function of the hash implementation. Figure 5.2a presents the (median) running time of this function for each input length. In broad terms, for inputs that are slightly shorter than a full block, the call to 'finalize' must execute an additional internal call to the hash compression function. In contrast with Figure 5.2b which shows the running time of the 'update' function; Figure 5.1 shows the total running time for computing HMAC-SHA256, which includes both functions as internal calls, and therefore shows both step-like behaviors.

We repeated the calculation from Subsection 5.13.1 for this smaller side channel. With 1,000 samples per case (2,000 measurements overall), and a 20% false positive rate, the false negative rate is 7.72%.[5] Increasing the number of measurements to 20,000 achieved a false positive rate of 0.004170% and a false negative rate of 0.012530%.[6] Repeating the same calculation from Subsection 5.13.1, the attack, therefore, requires roughly 302 million handshakes with the target server in total (compared to 22 million handshakes when exploiting the easier case of exploiting $\mathcal{O}^{\mathrm{H}}$, as described in Subsection 5.13.1). These numbers are not necessarily optimal.

5.13.3. Example of a Side Channel in OpenSSL

An example for a micro-architectural side channel that also affects protocols that do **not** strip leading zero bytes can be found in OpenSSL. OpenSSL has two functions to compute the shared secret of a Diffie-Hellman operation: `DH_compute_key_padded()` and `DH_compute_key()`. The big number library of OpenSSL does not preserve leading zero bytes, therefore `DH_compute_key_padded()` internally calls `DH_compute_key()` and then pads the leading zero bytes afterwards, creating a side channel (see Listing 1).

Other Classification Methods and Scenarios. Estimating the cost of performing the attack over the public Internet is an interesting challenge, but outside the scope of this work. Crosby et al. [108] have examined the feasibility of performing such timing attacks and found significant variability that depends on the attacker and victim hosts and the distance between them. They have also suggested a different classifier than the one we use, the *Box Test*. We have in fact, initially used this test as our classifier, but it significantly underperformed the Mann-Whitney test. Surprisingly, Crosby et al. have also considered the Mann-Whitney test, but reported that it underperformed their Box Test [108] (their test setup includes measurements on the same LAN, similarly to ours, as well as measurements over the Internet). The reason for this discrepancy is unclear to us. At any rate, providing a comprehensive comparison of classifiers is again an interesting task, but also out of scope for this work.

[5] As before, we estimated these rates using 200,000 experiments.

[6] We estimate these rates using 10 million sets of randomly-sampled 20,000 measurements. We further note that we configured the test with a false positive rate of 0.01%, but obtain a lower false positive rate. This could be an artifact of the approximations the test uses internally.

```
int DH_compute_key_padded(unsigned char *key, const BIGNUM *pub_key, DH *dh)
{
    int rv, pad;
#ifdef FIPS_MODULE
    rv = compute_key(key, pub_key, dh);
#else
    rv = dh->meth->compute_key(key, pub_key, dh);
#endif
    if (rv <= 0)
        return rv;
    pad = BN_num_bytes(dh->params.p) - rv;
    if (pad > 0) {
        memmove(key + pad, key, rv);
        memset(key, 0, pad);
    }
    return rv + pad;
}
```

Listing 1: DH shared key computation in OpenSSL 1.1.1g.

5.14. Solving the HNP

In order to show that we were actually able to solve the resulting equation systems, we simulated and solved the HNP problem for DH groups G with 1024, 1036, 2048, 3072, and 4096 bits and varying oracle sizes $k = 8, 16, 24$.

To reduce the number of exponentiations we do not simulate querying the oracle, because the oracle for larger bit leaks only has a very low success probability. In order to avoid a large number of false guesses, we *choose* values $0 < y'_i < 2^{n-k}$, which we interpret as a value $y'_i = g^{ab}g^{br_i}$ with $\mathrm{MSB}_k(g^{ab}g^{br_i} \bmod p) = 0$ for some unknown r_i. We acknowledge that this setup is slightly different to the Raccoon attack (as pointed out by Aldaya and Brumley [109]). We practically evaluated this concern but could not find a significant difference in the HNP computations. We then calculate $t_i := (g^{ab})^{-1}y'_i = g^{br_i} \bmod p$, $1 \le i \le d$, and assume that we could have guessed a corresponding r_i with probability $1/2^\ell$ in the first place. We take $y_1 = y_2 = \ldots = y_d = 2^{n-k-1}$ and get d equations $|g^{ab}t_i \bmod p - y_i| < p/2^{\ell+1}$, where $\ell = k - \epsilon$ is the effective number of bits leaked (see Section 5.11). To solve this instance of the HNP, we consider the lattice $\mathcal{L}(B)$ in $\mathbb{Z}^{d+2}$ generated by the column vectors of

$$B = \begin{pmatrix} 1 & & & & & \\ & \lceil p/2 \rceil & & & & \\ \lceil 2^\ell \rceil t_1 & \lceil 2^\ell \rceil y_1 & \lceil 2^\ell \rceil p & & & \\ \lceil 2^\ell \rceil t_2 & \lceil 2^\ell \rceil y_2 & & \lceil 2^\ell \rceil p & & \\ \vdots & \vdots & & & \ddots & \\ \lceil 2^\ell \rceil t_d & \lceil 2^\ell \rceil y_d & & & & \lceil 2^\ell \rceil p \end{pmatrix}.$$

This lattice contains the vector $v_1 = (p, 0, \ldots, 0)^\top$, which is usually a shortest non-zero vector. More importantly, the lattice contains the *hidden vector* $v_2 = (\alpha, *, \ldots, *)^\top$, where $\alpha = \min\{g^{ab} \bmod p, p - g^{ab} \bmod p\}$. The first component of that vector reveals the secret g^{ab}.

DH group n, ϵ	k		
	24	16	8
RFC 5114 $n = 1024, \epsilon = 0.532$	$\beta = 40$, $d = 50$ $T = 6\text{s} \pm 1\text{s}$	$\beta = 40$, $d = 80$ $T = 26\text{s} \pm 5\text{s}$	$\beta = 60$, $d = 200$ $T = 29881\text{s} \pm 26085\text{s}$
LibTomCrypt $n = 1036, \epsilon = 0.000$	$\beta = 40$, $d = 50$ $T = 5\text{s} \pm 1\text{s}$	$\beta = 40$, $d = 80$ $T = 24\text{s} \pm 6\text{s}$	$\beta = 60$, $d = 180$ $T = 6045\text{s} \pm 2101\text{s}$
SKIP $n = 2048, \epsilon = 0.056$	$\beta = 40$, $d = 100$ $T = 119\text{s} \pm 26\text{s}$	$\beta = 60$, $d = 160$ $T = 1417\text{s} \pm 136\text{s}$	
RFC 3526 $n = 3072, \epsilon = 0.000$	$\beta = 40$, $d = 150$ $T = 1120\text{s} \pm 96\text{s}$	$\beta = 60$, $d = 250$ $T = 32852\text{s} \pm 4356\text{s}$	
RFC 7919 $n = 4096, \epsilon = 0.000$	$\beta = 40$, $d = 200$ $T = 5373\text{s} \pm 355\text{s}$		

Table 5.4.: Our parameter choices and calculation costs to recover g^{ab} in a Raccoon attack for five well-known DH groups, using BKZ 2.0 with block size β, number of equations d and average calculation time T. We aborted the BKZ reductions as soon as the hidden number was found (up to BKZ loop completion). Each simulation was repeated 16 times with random secret on a vCPU with 2 GHz clock speed. The bit-size n of the modulus and its bias $\epsilon = n - \log_2(p)$ are also given. Note that for $k = 8$, we had to use more equations for the RFC 5114 group than for the LibTomCrypt group, mainly due to the larger bias ($\ell = 7.468 \ll 8$).

The expected length of the hidden vector v_2 is approximately $\sqrt{(d+2)/12} \cdot p$. But also, by the Gaussian heuristic, the length of a 'typical' shortest vector in $\mathcal{L}(B)$ is approximately $\sqrt{(d+2)/(2\pi e)}(\det B)^{1/(d+2)}$, where $\det B \approx 2^{\ell d - 1} p^{d+1}$. If the number d of equations is sufficiently large and ℓ is not too small, v_2 is expected to be smaller than typical shortest vectors in $\mathcal{L}(B)$ and we may hope to recover $\pm v_2$ as the second vector (after $\pm v_1$) in a reduced basis of $\mathcal{L}(B)$.

For our experiments, we used the BKZ 2.0 [110] version of the Block-Korkine-Zolotareff [111] lattice basis reduction algorithm with two block sizes $\beta = 40, 60$. We used the implementation of the fplll/fpylll library [112] in the SageMath [113] computer algebra system. The results are shown in Table 5.4.

5.15. Impact on TLS and Beyond

Exploitability. The Raccoon attack is generally hard to exploit since the prerequisites for the attack are quite rare nowadays. Parallel to the disclosure of this vulnerability, the last major browser (Firefox) stopped supporting DHE cipher suites. Even if the conditions of the attack are met, the attack still requires precise timing measurements, which are hard to perform in real networks.

Stronger attackers in a co-located setup may be able to use more advanced techniques like cache side channels to avoid timing measurements. We consider these stronger attacker models outside the scope of this work.

Attacking ECDH and ECDHE Cipher Suites. ECDH(E) cipher suites are generally not affected by the Raccoon attack, as TLS mandates that leading zero bytes are preserved. However, we identified some implementations which strip leading zero bytes from the coordinates, and then add those bytes back. This may result in a small timing side channel that leaks the MSB of the x-coordinate of the shared point. The EC-HNP [114] is related to the HNP and could potentially be applied here. However, a full analysis of this potential vulnerability is outside the scope of this work.

Downgrading TLS Sessions to DHE. Typical TLS connections are established with TLS ECDHE cipher suites. If an attacker can perform the complete attack within the handshake timeout, the attacker could perform a downgrade attack, and target TLS sessions that would otherwise not use Diffie-Hellman. The attacker acts as a MitM and removes any non-DHE cipher suites from the cipher suite list in the `ClientHello` message. Assuming both the client and server support at least one common DHE cipher suite, they will then attempt to handshake with it. The primary defense mechanism in TLS against such attacks is the `Finished` message, which includes a hash over the entire session transcript. But since the attacker learns the shared secret within the handshake timeout, the attacker can forge a valid `Finished` message, leading to a full break in security.

However, performing the attack fast enough is likely infeasible. The typical handshake timeout is around 30 seconds. The attacker needs to handshake with the victim server millions of times within this short period while performing accurate timing measurements for each server response. Furthermore, the attacker then needs to solve an instance of the HNP problem, which we were only able to accomplish with hours of computation time for small leaks.

One caveat is that some TLS libraries exhibit behavior that allows an attacker to stall TLS handshakes indefinitely [115]. Such behavior would make the online downgrade plausible.

Moreover, TLS False Start [71] allows the client to send encrypted application-layer data before receiving the server's `Finished` message. In principle, if the client is willing to use DHE with False Start, the attacker does not need to learn the shared secret within the handshake timeout but can do so at any point in the future. The data sent under False Start is typically particularly sensitive, such as authentication cookies; compromise of this data at any point in the (short) future typically leads to a full break in security. The False Start standard explicitly allows DHE cipher suites, but only with well-known groups with 3072-bit modulus or larger; however, typical TLS client implementations disallow DHE use with False Start altogether. To summarize, this concern is mostly theoretical.

TLS 1.3. In TLS 1.3 the leading zero bytes are preserved for DHE cipher suites (as well as for ECDHE ones). So broadly speaking, Raccoon does not apply to TLS 1.3. Note that our attack could work on a TLS 1.3 variant that explicitly allows key reuse (or even encourages it), called ETS or eTLS [116]. If ephemeral

keys get reused in either variant, they could lead to small side channels.

DTLS. The DTLS KDF [7] is identical to that of TLS, and has the same properties in regard to timing. However, an attacker may not be able to measure the timing difference, as DTLS does not necessarily send an error message when sessions are ungracefully terminated: DTLS is UDP-based, so it does not send TCP FIN or RST packets, and some implementations do not send alert messages at all. An attacker may be able to overcome these difficulties using techniques similar to Paterson et al. [117], but we consider this out of scope for this work.

SSH. In SSH, ephemeral key reuse is far less common than in TLS (as shown by Valenta et al. [118] in 2017). This is probably due to the more homogeneous deployment of SSH, and the raised security requirements, as a break in SSH could lead to remote code execution. In SSH, the shared secret is encoded as an `mpint`, which explicitly removes leading zero bytes. In contrast to TLS, the shared secret is hashed with the session transcript to generate the 'exchange hash'. This essentially removes the attack prerequisite for a dangerous modulus size. The attacker can guarantee that the difference of a stripped zero byte always results in less processed blocks within the hash function, since attacker-controlled messages with non-fixed length are included in the computation. To summarize, SSH servers which reuse ephemeral DH values are at greater risk to the attack than TLS servers which do so, but such SSH servers are rare.

Interestingly, SSH also strips the leading zero bytes of the shared secret for X25519. RFC 8731 [119] explicitly mentions that this is a potential problem, as it leaks the leading zero byte of the shared secret, but decided not to address the issue for backwards compatibility reasons.

> *The way the derived binary secret string is encoded into a mpint before it is hashed (i.e., adding or removing zero-bytes for encoding) raises the potential for a side channel attack which could determine the length of what is hashed. This would leak the most significant bit of the derived secret, and/or allow detection of when the most significant bytes are zero. For backwards compatibility reasons it was decided not to address this potential problem.*

Hybrid Key Exchange in TLS. Hybrid key exchange refers to the practice of using two key exchange algorithms in parallel. The primary motivation is to use both a classical key exchange algorithm and a post-quantum one, in hope of achieving security as long as at least one of the algorithms is not broken. As of the initial publication of the Raccoon attack, preliminary work by the IETF on such designs centered on an Internet Draft, 'Hybrid key exchange in TLS 1.3' [120]. The original language in that document allowed for variable-length secrets, should the need arise in the future. [7] Shortly after the Raccoon attack was originally published, we have reached out to the IETF regarding this

[7] However, the document noted that all Round 2 candidates for post-quantum key exchange have fixed length for the shared secret.

issue. In response to our findings, the document no longer allows for variable-length secrets, and the security consideration section now notes the danger in using such secrets.

Other Protocols. XML Encryption [121] and IPsec [122, Section 2.14] preserve leading zero bytes. JSON Web Encryption [123] only offers ECDH key agreement. The established shared secret is processed according to NIST SP 800-56A [124], which requires leading zero bytes to be preserved.

5.16. Conclusion

Beyond the specifics of the attack, we argue that its existence can also teach us broader lessons for cryptographic protocols.

Forgoing Forward Secrecy is Dangerous. Forward secrecy is a well-known security goal for cryptographic protocols and was intensively analyzed in the context of TLS by Springall et al [106]. Our attack exploits the fact that servers may reuse the secret DH exponent for many sessions, thus are forgoing forward secrecy. In this context, Raccoon teaches a lesson for protocol security: For protocols where some cryptographic secrets can be continuously queried by one of the parties, the attack surface is made broader. The Raccoon attack shows that we should be careful when giving attackers access to such queries.

Secrets should be Constant-size. The dangers of non-constant-time implementations are well-known. For example, they have been repeatedly demonstrated to break ECDSA as used in TLS. One of the reasons for these breaks is that the processing of variable-length secret values within the implementation usually results in non-constant execution time. We argue that future protocol designs should make sure that all their secrets (including intermediate values and their internal number representation) are of fixed size.

Countermeasures. The most straightforward mitigation against the attack is to remove support for TLS-DH(E) entirely, as most major client implementations have already stopped supporting them. Moreover, server operators should disable DHE key reuse, which completely prevents the practical attack even if support for DHE cipher suites is prioritized.

Updating the TLS specification to preserve leading zero bytes is impractical. In fact, experience shows that even deployment of new protocol features conforming to the existing specification is hard. This is because many implementations in the wild only implement a subset of the specification, often in a buggy way. David Benjamin presented a talk at the RWC 2018 on the issue [125]. Since TLS 1.3 is not vulnerable to the attack by design, we propose to focus on the TLS 1.3 deployment rather than trying to update the old specification.

To prevent timing-based side channels in legacy applications with length-varying secrets, vendors must ensure that functions are implemented in constant time. This can be done as in the Lucky 13 mitigation or by computing

the values for different fake parameters and discarding the fake ones afterwards. However, one has to be very careful when implementing such mitigations; previous research has shown that this kind of mitigations adds code complexity and may still leave the side channel open [85] or introduce even more severe vulnerabilities [15].

Interoperability should not Dictate Cryptographic Design. While TLS as it is standardized encourages non-constant-time implementation, it is worthwhile examining just how this came to be. The first wide-scale Diffie-Hellman deployment was likely SSLv3 [8], but Internet standards from that era are not well-specified. The SSLv3 standard itself [126] does not specify whether leading zero bytes should be stripped.

PKCS #3 [127], the 'Diffie-Hellman Key-Agreement Standard', mandates that leading zeroes be preserved. This is also the case in RFC 2631 [128], 'Diffie-Hellman Key Agreement Method'. However, according to [129], some implementations have misread the standards and stripped leading zero bytes, leading other implementations to follow suite so that they could interoperate. By the time this was fully identified, changing every SSLv3 deployment at once was obviously impractical, so this misfeature persisted. Moreover, this behavior had to be standardized in the TLS 1.0 RFC [54], so that future implementations would interoperate with existing ones. Therefore, not enforcing the relevant standards in the name of interoperability came at the cost of a latent vulnerability, discovered almost 20 years later. A similar observation can be seen with design ob TLS 1.3, where a previously well engineered protocol was disfigured to be deployable alongside buggy implementations.

Modal Behavior in Crypto Implementations is Dangerous. The existence of direct oracles for leading zero bytes is perhaps surprising at first glance. Naïvely, stripping zero bytes is a simple, negligible part of the KDF computation; there is no reason to expect this code to cause an implementation to emit different responses to later, invalid `Finished` messages. However, as a similar example, we note that some versions of SChannel, the Microsoft TLS implementation, do not strip leading zero bytes, and therefore fail to handshake with other implementations in roughly 1/256 of DH connections [130, 131] [9]. A similar bug was identified and fixed in the JDK [132]. If such behaviors are found in the wild even when connections exhibit detectable failures, it is not surprising that implementations exhibit similarly obscure behaviors in edge cases. Whether these bugs lead to a break in security then depends on the precise behavior caused by each bug.

This finding therefore demonstrates again that when cryptographic implementations need to include rarely-used code paths, these paths are often not well-tested, and may lead to practical attacks, as was already demonstrated by Juraj Somorovsky [15] and Böck et al [89]. Cryptographic standards should

[8] SSLv2 does not allow use of Diffie-Hellman.

[9] We note that this bug likely does not harm security, it merely causes some small portion of handshakes to fail.

therefore take into account the implementation complexity, and in particular the typical number of required code paths.

6 Alpaca Attack

This chapter introduces the Alpaca attack [10], a series of cross-protocol attacks on TLS that abuses subtle flaws in the authentication of TLS. The original paper takes a deep dive into the world of cross-protocol attacks by exploring the impact of the vulnerability in the context of SMTP, IMAP, POP3, and FTP. During my dissertation, I co-authored the attack. While I did not work on the actual exploits of the attack, I worked with Marcus Brinkmann on the generic cross-protocol attack on TLS, without consideration for the application layer protocol. Additionally, I worked on the constraints of the attack in regards to TLS, and TLS-based countermeasures. In regards to practical analysis, I assisted Marcus Brinkmann during the evaluation of the Internet-wide scans. Since my contributions to the attack are rather well contained, this chapter only introduces the aspects of the Alpaca attack I have personally worked on. For a complete description of the attack and the exploits, I refer to the original publication [10]. The results of the Internet-wide scans are presented in Section 9.2.

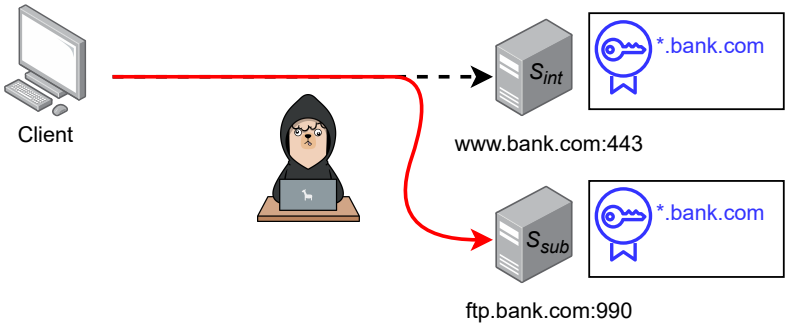


Figure 6.1.: Sketch of the generic Alpaca attack. A client wants to connect to S_{int} but the attacker forwards the traffic to S_{sub}. As long as S_{sub} has a certificate that could also be valid for S_{int}, it is possible that the modifications done by the attacker are not noticed by either S_{sub} or the client.

6.1. Generic TLS Cross-protocol Attack

A generic cross-protocol attack involves a client C and two application servers S_{int} and S_{sub}. The client C uses protocol A with the *intended* server S_{int}. The *substitute* server S_{sub} uses an unrelated protocol B and runs on a different TCP endpoint (IP, port). However, S_{sub} has a certificate that is *compatible* with S_{int}, i.e., the certificate could be used by S_{int} in place of its regular certificate without breaking the intended connection from the client. The goal of the attacker is to trick either S_{sub} into accepting application data from C or to trick C into accepting application data from S_{sub}. Because C and S_{sub} use different protocols, this type of attack is called a *cross-protocol* attack. The attack works like this:

1. The MitM attacker interposes the TCP connection between C and S_{int}, and forwards all data from C to S_{sub} and vice versa. Optionally, the MitM first creates a new TLS endpoint on S_{sub} with STARTTLS and only then forwards the TLS traffic from C.

2. The application server S_{sub} performs the TLS handshake with the client C and presents its certificate Cert_{sub}.

3. Because Cert_{sub} is compatible with S_{int}, the client C accepts it and completes the TLS handshake. Subsequently, C will send application data to S_{sub}.

4. S_{sub} tries to interpret the data sent by C. Because the client sends application data in format A and the S_{sub} expects format B, this may result in security violations.

5. If S_{sub} responds by sending, e.g., error messages to C in format B, the client processes these which may also result in security violations.

The presented attack breaks the authentication of the connection in step 3, when C finishes the handshake with S_{sub}, as C is not noticing that it performed the handshake with S_{sub} instead of S_{int}. In this case we say that S_{int} and S_{sub} are *vulnerable to cross-protocol attacks*. This loss of authentication can lead to severe security issues at the application layer and potentially a loss in confidentiality; for example, if S_{sub} writes application data to a log file readable by the attacker. In practice, cross-protocol attacks are hindered by a series of obstacles. We identify the following requirements for cross-protocol attacks to be exploitable:

- **TLS Compatibility.** The client C and application server S_{sub} must complete the TLS handshake, and C must accept the certificate of the substitute server as valid for the intended server. We provide more details on this requirement in Section 6.2.

- **Tolerance To Protocol Embedding.** S_{sub} should tolerate a certain amount of invalid traffic that comes from protocol A in which the payload

for S_{sub} in format B is embedded. Likewise, if the client is expected to process any response by S_{sub}, it should tolerate a certain amount of invalid traffic that comes from protocol B in which the payload for C in format A is embedded.

- **Application Server Exploitability.** S_{sub} must provide some feature, mechanism, or behavior supporting the attack. The details are protocol- and implementation-specific, but generally, the exploited behavior will be unexpected by C and differ considerably from the behavior of S_{int}, resulting in some form of security violation.

6.2. TLS Compatibility

To enable cross-protocol attacks, the server S_{sub} must provide a certificate and TLS configuration that is compatible with the certificate and configuration of S_{int} such that C will successfully complete a handshake with S_{sub} in a MitM scenario. We will now describe the most important requirements for this.

Certificate Names. The client application must accept the server names in the certificate of S_{sub} as valid for S_{int}. This can be the case for one of two reasons: 1. The certificate presented by S_{sub} has the hostname of S_{int} in the CN field or SAN extension, or 2. The certificate of S_{sub} is a wildcard certificate that matches the hostname of S_{int} (e.g., `*.bank.com` matches `www.bank.com`). Such configurations occur spontaneously when an administrator, unaware of the risk of cross-protocol attacks, deploys a multi-domain or wildcard certificate to save costs and administrative effort, or simply copies a web server certificate to another application server to support opportunistic encryption without validation, as is common for non-HTTPS services [133]. Note that for the success of cross-protocol attacks it is not required that the certificate presented by S_{sub} is valid for S_{sub} itself.

Certificate Validity. The certificate must also satisfy a broad range of other conditions to be considered valid by C. Most importantly, the certificate must be signed by a certificate authority trusted by C, and the certificate must not be expired.

Note that there is no possibility to define the designated application layer protocol in an X.509 certificate. While a certificate can include an extended key usage extension, this extension can only indicate a very broad purpose for which the certified public key may be used [59], for example, code signing, client/server authentication, or email protection.

TLS Handshake Parameters. The TLS protocol itself can also cause the attack to fail. In particular, the client C must support at least one TLS version and cipher suite offered by S_{sub} which may differ from those provided by S_{int}. This is especially important if the two protocols adopt new versions, cipher suites, and extensions at different speeds, for example, if the client C deprecates some features before S_{sub} is updated to support suitable replacements.

ALPN. If the client C supports the ALPN extension, it will send only the identifiers for the intended protocols. If S_{sub} is also implementing the ALPN extension, it cannot choose a matching application layer protocol. In this case, the ALPN extension mandates to abort the handshake and send a fatal TLS alert message [66]. Thus, at first glance, this might prevent the attack. However, ALPN was never considered a security feature, but merely a mechanism to multiplex different protocols on the same TCP endpoint [66]. If the substitute server is unaware of ALPN, or the ALPN extension is not transmitted by C, or a failure in ALPN negotiation is silently ignored by the server, the handshake proceeds despite ALPN. We will show in Subsection 9.2.2 that this is often the case for servers implementing protocols other than HTTPS.

SNI. A client may specify the server name in the SNI extension. If S_{sub} is not responsible for resources on the indicated server name, it can reject the connection and thus prevent the attack. However, cross-protocol attacks are not affected by SNI if two services run under the same name, if the substituted server does not implement it, or if the server is misconfigured. Similar to ALPN, SNI was not specified as a security feature, but to multiplex different virtual servers (possibly implementing the same protocol) on the same TCP endpoint. It is still not widely supported outside of HTTP. We will show in Subsection 9.2.2 that servers implementing protocols other than HTTP often ignore the SNI extension.

6.3. Countermeasures

Countermeasures at the Application Layer. Previous efforts to stop cross-protocol attacks tried to mitigate the issue at the application layer, for example, by closing the connection if HTTP is detected instead of a valid command. From a practical point of view, it is unreasonable to expect implementers to be aware of all (including future) possible cross-protocol attacks and defend against them one by one.

While such measures can potentially stop the exploitation of individual protocol confusions, they cannot generally stop the attack. Whenever a client finishes the handshake with S_{sub}, the authentication as promised by TLS has already been broken. At this point, no application data has been exchanged yet, therefore no application layer countermeasure can prevent the general cross-protocol attack.

Countermeasures with TLS Certificates. A common proposal is to use different (incompatible) certificates for different service endpoints. However, enforcing such a policy is challenging in practice. Certificate validation is limited to hostnames; thus, each service would have to be hosted on a unique subdomain. Furthermore, no certificate should be issued for more than one hostname, which effectively prohibits the use of wildcard certificates. However, the very common use of wildcard certificates in practice shows that they provide significant value

to administrators. Even strict certificate exclusivity does not prevent all possible attacks. The attacker could still steal the cookie using a service hosted on a subdomain or perform session fixation attacks.

Another idea would be to define different certificate usages for distinct services. While the X.509 standard defines the extended key usage extension [59], this extension only allows distinguishing TLS server certificates from those used for email signing, IPsec, or OCSP, and does not provide a mechanism to authenticate the application protocol on top of TLS.

We conclude that the required organizational and behavioral changes to achieve certificate exclusivity are so large that they can only be considered a long-term countermeasure.

ALPN Mitigates All Cross-Protocol Attacks. In 2015, Horn suggested the use of ALPN by protocol designers to mitigate cross-protocol attacks. We now describe how this countermeasure can be implemented in a backward-compatible way. If ALPN is supported by both client and server, the standard requires that the connection is closed if no common protocol can be negotiated. This strict implementation mitigates all cross-protocol attacks because a client and the substitute server implementing a different protocol than the client will never complete a TLS handshake.

Today, we see different levels of ALPN support deployed. For HTTPS, all major clients already implement ALPN to support HTTP/2, so deploying ALPN in exploitable application servers will prevent our attacks on HTTPS. In an Internet-wide scan, we found that 72.5% of HTTPS servers already support ALPN. Although this is promising, we also found that less than 1.3% terminate the connection if no protocol can be negotiated. We have also shown in our scans that virtually all SMTP, IMAP, POP3, and FTP servers do not support ALPN or do not terminate if no protocol can be negotiated.

As a path forward, we propose that initially, servers start to implement ALPN strictly according to the standard, so connections created by clients sending the ALPN extension (i.e., browsers) are protected from exploitation. In parallel, clients for all protocols (SMTP, IMAP, POP3, and FTP) can be upgraded to send the ALPN extensions. Migrating to this secure configuration is easy and backward-compatible, as the clients and servers can independently enable the extension on their respective sides at some convenient time, while still accepting legacy connections. Once a client and an application server have *both* enabled ALPN, that particular server can no longer be exploited to attack connections by that client to other, vulnerable servers in the network.

Eventually, clients and servers may choose to require the use of ALPN by the other side, at the cost of breaking backward compatibility with legacy implementations.

Countermeasures with SNI. In the same way the ALPN extension protects against cross-protocol attacks, the SNI extension can protect against cross-hostname attacks if it is implemented strictly (ie., the connection is terminated if no matching host is found), which is allowed by the standard. This can protect

against cross-protocol attacks where the intended and substitute server have different hostnames, but also against some same-protocol attacks such as HTTPS virtual host confusion [134] or context confusion attacks [135].

Unfortunately, some servers are currently not entirely aware of the hostnames they are responsible for. Adding a strict SNI validation to those servers can cause connections to break if hostnames are missing or clients are misconfigured. Still, we recommend enabling strict SNI checking, particularly for new configurations.

Same-Host, Same-Protocol, Cross-Port Attacks. Even with strict ALPN and SNI implementations, we still face potential confusion attacks when the intended and substitution server have the same hostname, implement the same protocol, but run on different ports. These *cross-port* attacks can currently not yet be mitigated at the TLS layer, because there is no way for the client to communicate the intended port number to the server. Defining such a feature, for example as a new TLS extension, is certainly possible, but would require an upgrade to all TLS libraries and applications.

6.4. Conclusion

We demonstrated that the lack of strong authentication of service endpoints in TLS can be abused by attackers to perform powerful cross-protocol attacks with unforeseeable consequences. Existing countermeasures are ineffective because they do not address all possible attack scenarios.

We have identified one countermeasure that is far superior to others: the pervasive use of the ALPN extension to TLS by both client and server. Luckily, ALPN is easy to deploy with the next software update without affecting legacy clients or servers.

In a broader sense, the Alpaca attack demonstrates yet again that all cryptographic measures, when applied to real-world applications, should be bound to the context of their legitimate use to prevent confusion attacks on the protected content. Binding the TLS connection to a specific application layer protocol allows peers to protect themselves against any known and unknown cross-protocol attack. SNI can extend this protection to same-protocol attacks on different hostnames. These countermeasures make sure that a message for the intended protocol is not mistaken for a message in the substituted protocol, as demanded by the Rule of Thumb 5 [136] for safeguarding authentication protocols against environmental threats. However, we have also seen that services that share the same hostname and protocol can not be protected against confusion attacks by existing TLS standards.

7 TLS-Attacker

Analyzing TLS implementations can be complex. In contrast to other protocols like HTTP, SMTP, POP3, or FTP, the TLS protocol is a binary protocol making it hard to read and write without special tooling. Additionally, due to the cryptographic nature of the protocol, the messages usually depend upon each other and are often encrypted, making hard coded messages unpractical. Therefore, to analyze TLS libraries dynamically, the community started writing small scripts or tools which could perform the desired connections. For this purpose, it was common that real TLS libraries were used or manipulated to send and receive partially invalid messages. While this approach works to some extent, it resulted in many works always starting from scratch or spending tremendous amounts of time manipulating a real-world implementation to achieve their goals. This is an often lengthy process as real-world libraries are not designed to do this. This process shifted when frameworks were introduced to ease the development of analysis tools. The first major framework of this kind was FlexTLS [137] in 2015, which is based on miTLS [138], a verified TLS implementation. FlexTLS is one of the first frameworks that allowed the transmission of out-of-order messages and was used by Beurdouche et al. [101] for the FREAK attack.

In 2016, Juraj Somorvosky presented TLS-Attacker [15], a flexible framework designed to test TLS libraries. Since around 2017, I took over the project and was responsible for its development while Juraj Somorovsky was supervising me. I drastically extended the framework and introduced many new concepts that allowed the framework to grow. The framework was developed in joint work with my colleagues, employed students, and countless students who contributed to the project during their Master's or Bachelor's theses under my supervision. When I joined the project, TLS-Attacker 1.2 was the most recent version. The first version containing major architecture changes that I contributed start with TLS-Attacker 2.0. At the time of writing, the current version of TLS-Attacker is 4.8. This section presents the most recent version of the framework.

7.1. Core Concepts

TLS-Attacker is a framework that allows a security tester or developer to quickly test any feature of a TLS implementation or to prototype attacks very quickly. To achieve this purpose, TLS-Attacker needs to implement all relevant TLS RFCs and make its internal computations accessible such that the user can influence them. TLS-Attacker achieves this by implementing the whole TLS protocol stack with low-level Java code (besides cryptographic primitives like AES, or hash functions).

ModifibaleVariables. To deal with dynamic messages that depend on each other, TLS-Attacker computes all messages at runtime. To enable flexible access to the message contents, every variable within the TLS stack is implemented as a *ModifiableVariable*. A *ModifiableVariable* functions as a standard variable but allows the user to place a modification within it. A modification is a small program hook executed whenever the value is accessed. Consider the example in Listing 2.

```
ModifiableInteger i = new ModifiableInteger();
i.setOriginalValue(30);
System.out.println(i.getValue()); // 30
i.setModification(new IntegerAddModification(20));
System.out.println(i.getValue()); // 50
```

Listing 2: Example code on how to use `ModifiableVariables`.

Here, a developer creates a new *ModifiableInteger*, and sets its value to 30. When the user calls the `getValue()` function, the function returns the 30 back. If the user then sets a modification this *ModifiableVariable* (in this case an `add 20` modification), the modifiable variable return $30 + 20 = 50$ from now on. The *ModifiableVariable* concept is developed in its own repository.[1] An overview of the available *ModifiableVariables* and their modifications is given in Table 7.1.

Chooser. Another core idea behind TLS-Attacker is that any protocol flow or modification, no matter how unreasonable, should be able to be executed by the framework. That is, even if it does not seem to make sense at first glance, any instructions given to TLS-Attacker should be executed and rarely result in an inability of TLS-Attacker to proceed with the execution. This can be tricky, as TLS messages depend upon each other and rely on specific values being available. TLS-Attacker solves this by introducing a `Context` and a `Config`. The `Context` contains all the values established within a connection at runtime, for example, which cipher suites were supported, which version was negotiated, and so on. In comparison, all values that are not determined at runtime or might be missing at runtime have entries (and default values) in the `Config`. If possible, TLS-Attacker will always try to create messages at runtime that are

[1] `https://github.com/tls-attacker/modifiableVariable`

Type	Operation	Description
BigInteger	Add	Adds a given number to the original value
	Explicit	Overwrite the original value with a provided number
	Interactive	Asks the user from *stdin* to provide a number
	Multiply	Multiplies the original number with a given number
	ShiftLeft	Shifts the bit representation of the BigInteger to the left by a certain amount
	ShiftRight	Shifts the bit representation of the BigInteger to the right by a certain amount
	Subtract	Subtracts a given number from the original value
	XOR	XOR's the original value in a bit representation with a provided value starting at a provided position
Boolean	Explicit	Overwrites an original value with a provided value
	Toggle	Toggles a given original value
Byte	Add	Adds a given number to the original value
	Explicit	Overwrite the original value with a provided byte
	Subtract	Subtracts a given number from the original value
	XOR	XOR's the original value in a bit representation with a provided value
ByteArray	Delete	Deletes a given amount of bytes from a byte array at a provided position
	Duplicate	Duplicates a given original value
	Explicit	Overwrites the original value with a provided byte array
	Insert	Inserts a byte array into the original value at a given position
	Shuffle	Shuffles the bytes in the original value by a given shuffling key
	XOR	XOR's the original value with a provided value starting at a provided position
Integer	Add	Adds a given number to the original value
	Explicit	Overwrite the original value with a provided number
	ShiftLeft	Shifts the bit representation of the Integer to the left by a certain amount
	ShiftRight	Shifts the bit representation of the Integer to the right by a certain amount
	Subtract	Subtracts a given number from the original value
	XOR	XOR's the original value in a bit representation with a provided value starting at a provided position
String	Explicit	Overwrite the original value with a provided string

Table 7.1.: Overview of all available modifications in the ModifiableVariable package.

as close to the specification as possible. TLS-Attacker will always prefer values from the `Context` for its message creation, but if these values are missing, fall back to the `Config` to retrieve a default value. The component responsible for this fallback is called `Chooser`. TLS-Attacker has a *reasonable* default `Config` which can be changed by the user. To illustrate concept, consider TLS-Attacker sending a `ChangeCipherSpec` and `Finished` message, without sending sending a `ClientKeyExchange` message before. The `ChangeCipherSpec` message instructs TLS-Attacker to encrypt the `Finished` message, but no `master secret` was computed yet. In this case, the `Context` has no value yet, and the `Chooser` will use the `master secret` from the `Config`, which contains an all zero default value, such that the `Finished` message can be encrypted.

Action System. To send and receive messages with TLS-Attacker, TLS-Attacker uses an *action system*. This system requires that the user defines a static `WorkflowTrace` that contains a list of actions (think commands) that TLS-Attacker executes one by one to perform a handshake. The most simple actions are `Send` and `Receive` actions. As their name implies, a `Send` action sends a list of protocol messages, while a `Receive` action tries to receive specific messages. With these two actions alone, it is already possible to create a `WorkflowTrace` that can perform a TLS handshake by sending the appropriate messages, and waiting for answers from the peer at the correct time. An example of a TLS-Handshake is given in Listing 3. The messages within the `WorkflowTrace` can have modifiable variables attached to them to modify their contents, while their values are computed at runtime with the assistance of the `Chooser`. TLS-Attacker implements over 60 actions that can be used to perform very complex handshakes or to control nuances during the handshake. After the execution of a `WorkflowTrace`, TLS-Attacker can check if the execution of the `WorkflowTrace` was successful (e.g., if all messages could be transmitted and all desired messages were received), which allows for convenient automation.

Behavior Options. The action system is already compelling but still leaves some questions. For example, what is TLS-Attacker supposed to do when the desired messages were not received during the execution? For this purpose, TLS-Attacker introduces different actions and configuration options in the `Config` which control how TLS-Attacker behaves. For example, the option `quickReceive` tells TLS-Attacker do not wait the whole timeout for further messages once all expected messages were already received to improve the performance of the connection.

XML Interface. To control TLS-Attacker without writing code and to store `WorkflowTraces` and `Configs` TLS-Attacker provides an XML interface. This enables quick tests without writing Java code and is also beneficial for exporting test cases that were automatically generated. An example of a serialized `WorkflowTrace` is presented in Listing 3. An example of a serialized `Config` is presented in Listing 4.

```
<workflowTrace>
    <Send>
        <messages>
            <ClientHello/>
        </messages>
    </Send>
    <Receive>
        <expectedMessages>
            <ServerHello/>
            <Certificate/>
            <ServerHelloDone/>
        </expectedMessages>
    </Receive>
    <Send>
        <messages>
            <RSAClientKeyExchange/>
            <ChangeCipherSpec/>
            <Finished/>
        </messages>
    </Send>
    <Receive>
        <expectedMessages>
            <ChangeCipherSpec/>
            <Finished/>
        </expectedMessages>
    </Receive>
</workflowTrace>
```

Listing 3: Example of a `WorkflowTrace` containing `Send` and `Receive` actions.

```
<config>
    <defaultSelectedNamedGroup>ECDH_X25519</defaultSelectedNamedGroup>
    <defaultSelectedCipherSuite>TLS_AES_128_GCM_SHA256</defaultSelectedCipherSuite>
</config>
```

Listing 4: Example of a `Config` that overwrites the default selected cipher suite and named group.

Supported Features. An overview of supported features is given in Chapter A.

7.2. Message Processing

To enable so many different features, it is essential that every feature follows a strict code flow and that different components of a feature are as isolated as possible such that the development of a new feature does not slow down the development of other features in the future. For this purpose, TLS-Attacker has a precise message processing flow, which is identical for every message. The flow is implemented through four components for every message within TLS-Attacker.

Parser. The parser is a class that translates a byte sequence into a Java object representing the message. The object then has getters and setters to access the individual fields of a message.

Serializer. Serializers are the counterpart to parsers, as they encode an object back to its byte sequence representation.

Preparator. All messages within a TLS-Attacker connection are computed at runtime. The preparator performs these computations. If noteworthy cryptographic computations are made, the preparator writes intermediate values to metadata *ModifiableVariables* within the message, such that the user can influence these computations with modifications. If computations have to be performed upon receiving a message, the preparator also executes these computations.

Handler. The handler is responsible for updating the `Context` with the contents of the message it is passed. For example, if we send or receive a `ClientHello`, we have to set the client random in `Context`, such that future messages can depend upon the `ClientHello`.

The process of sending a message is visualized in Figure 7.1. It starts with an empty message object (that may contain modifications on its *ModifiableVariables*). First, the values of the message are computed by the preparator at runtime. Then, the handler looks at the message and copies relevant values from the message inside the context, such that future messages can depend on the currently processed message. After that, the serializer translates the message into bytes which can then be passed to the next layer (i.e., the record layer).

Receiving a message is similar to the sending of messages. First, the received bytes are translated into an object by the parser class. After that, the preparator gets a chance to perform additional computations on the received data. The newly computed values are then stored as metadata alongside the message. At last, the handler receives the message and copies necessary values into the `Context` of the connection.

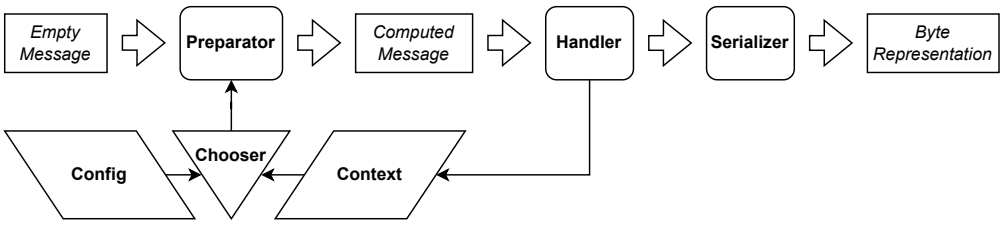


Figure 7.1.: Visualization of the message sending procedure of TLS-Attacker. In the beginning, messages are empty besides optional modifications. The preparator then computes the message contents at runtime using the `Chooser` to select appropriate values from either the `Context` or `Config`. The created message is then passed to the handler to update the context. After that, the message is serialized.

7.3. Man-in-the-Middle Attacks

A common task for TLS-Attacker is the development of MitM attack prototypes. To support this, TLS-Attacker allows `WorkflowTraces` with more than two parties. Each connection has to pass through TLS-Attacker, where each connection 'to' and 'from' the attacker gets assigned an alias. For each alias, TLS-Attacker creates a `Context` and separates the negotiated parameters for each connection. A `State` wraps every context, as well as the `WorkflowTrace` up. If more than one alias is defined, most actions require that an alias is assigned to them to specify in which `Context` they are executed. Some actions, like the `Wait` action, do not require an alias, as they apply globally. TLS-Attacker also offers multi-`Context` actions which are specifically designed for MitM attacks. For example, the `ForwardRecords` action receives records in the `Context` of one alias, and sends them (without trying to decrypt/encrypt them) in the `Context` of a different alias. As with all actions, these actions can also be serialized in a `WorkflowTrace`.

7.4. ASN.1-Tool and X.509-Attacker

ASN.1 and X.509 play a tremendous role within TLS, as they are used during authentication from both the client and the server to verify the peer's identity. While these standards are not TLS exclusive, TLS is the most common application. These standards introduce much complexity to the protocol, as they follow their own rules and encodings that differ from TLS. To allow a user access to this aspect of the TLS protocol, we developed ASN.1-Tool[2] and X.509-Attacker,[3] two tools allowing users to specify arbitrary TLS certificates similar to `WorkflowTraces` in TLS-Attacker. These custom certificates can then be used inside of TLS-Attacker.

[2] `https://github.com/tls-attacker/ASN.1-Tool`
[3] `https://github.com/tls-attacker/X509-Attacker`

7.5. Elliptic Curve Computations

Typical elliptic curve implementations do not allow computations with invalid values, for example, points that are not on the curve. TLS-Attacker faced a similar problem, as off-the-shelf elliptic curve libraries like JSSE or BouncyCastle do not allow these computations either. To deal with this problem, TLS-Attacker implements our own elliptic curve library that allows for otherwise invalid computations. If an 'undefined' computation should be performed, this library returns 'some' value instead.

7.6. Timing Attacks

It is a common goal for TLS analysis tools to analyze potential timing attacks. For this purpose, it is essential to accurately measure the time it takes for a peer to respond to a message. This turns out to be a non-trivial issue if one wants to avoid noise in the measurements, as the JVM, the operating system, and the network card introduce avoidable noise. We developed a proxy-based measuring setup visualized in Figure 7.2 to deal with this issue. Instead of measuring the time itself, TLS-Attacker forwards the data it wants to transmit to a proxy application. This proxy application then performs the actual timing measurements and forwards the response and the measured time back to TLS-Attacker. This allows the researcher to separate the measuring setup and the regular working machine, which can ease the prototyping process of new attacks. Additionally, the proxy can then use native code, allowing measurements closer to the actual hardware. The Timing-Proxy[4] of TLS-Attacker implements three different measuring techniques.

CPU. This approach uses CPU timestamps and the RDTSCP instruction in the `TimeSources` class to perform the timing measurements.

PCAP. This approach uses *libpcaps* internal timestamp to measure the timings of the packets.

Kernel. In this approach, the timing measurements are directly performed on the network card. The network card is instructed to attach hardware timestamps to each incoming packet. The accuracy of this approach is highly dependent on the quality of the network card. Network cards typically can only assign timestamps in discrete time frames. The resolution of the timestamps, therefore, depends on the clock frequency of the network card. The resolution is usually not very good for consumer network cards, but special hardware enables high-resolution time stamping. If such a card is available, this method is the most accurate method to perform remote timing attacks.

[4] `https://github.com/tls-attacker/Timing-Proxy`

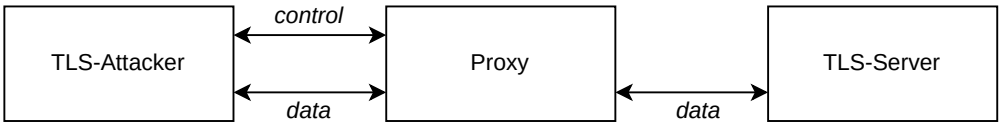


Figure 7.2.: An overview of the TLS-Attacker timing attack setup. TLS-Attacker controls the timing proxy with a control channel and then sends the data it wants to transmit to the server via a designated data channel. The measured timing can be retrieved with the control channel.

7.7. Applications

TLS-Attacker has a few example applications that show how to use TLS-Attacker as a software library and function as convenient tools for the analyst. After building TLS-Attacker, these applications are within the 'apps' folder.

TLS-Client.jar. TLS-Client.jar implements a simple TLS-Client that can execute `WorkflowTraces` from an XML interface or generate `WorkflowTraces` from a given *Config*.

TLS-Server.jar. Similar to TLS-Client.jar, TLS-Server.jar implements a simple TLS-Server. Once started, the server waits until a client connects and performs the configured connection.

TLS-Mitm.jar. In order to execute MitM attacks TLS-Mitm.jar can be used. To use the tool, the `WorkflowTrace` has to define the aliases for the execution and the actions have to define an alias as well (see Section 7.3).

TraceTool.jar. Creating XML `WorkflowTraces` can be tedious, especially for new users. For this purpose, TraceTool.jar was created. This tool functions similarly to TLS-Client or TLS-Server, but instead of connecting with the defined parameters, it outputs the XML for the created `WorkflowTrace`.

7.8. Sockets

While the TLS-Attacker `WorkflowTrace` concept is beneficial for the analysis of TLS implementations, it may also be desirable to test applications on the application layer with specific TLS parameters. For example, the 'Token-binding Protocol' [139] depends on the presence of the extended master secret (EMS) extension and the *RenegotiationInfo* extension and requires access to the exporter master secret. While testing applications with `WorkflowTraces` is certainly possible, the standard APIs of TLS-Attacker make it very tedious to develop applications on top of it. To bypass this limitation, TLS-Attacker also offers a socket API. Here, the user can either pre-execute the handshake

or provide a `Config` to TLS-Attacker with which TLS-Attacker should perform the handshake. After that, TLS-Attacker provides streaming-based access to the application layer, similar to a traditional socket-based TLS implementation.

7.9. External Template

As presented, TLS-Attacker on its own is already a powerful tool for the analysis of TLS-Implementations. However, its real strength first becomes visible when TLS-Attacker is used as a software library to build even more powerful analysis tools that further automate the analysis. In sections 8.1, 8.4, 10 and 11 we introduce tools that do exactly that. To ease the development of such tools, a Java template project[5] exists that can be adapted for custom tools.

7.10. TLS-Docker Library

Analyzing TLS implementations in an academic context frequently requires that different TLS implementations are compiled, configured, and started alongside each other. This can be tedious, especially when working with multiple machines during collaborations with multiple people. Additionally, the different TLS implementations may be required in different versions, which can require conflicting dependencies installed alongside each other. To ease the automatic evaluation of TLS libraries, we created the TLS-Docker-Library,[6] a collection of TLS implementations (for clients and servers) in docker images with a Java API that abstracts away from the individual command line flags of the implementations. An overview of the available images is given in Table 7.2. Listing 5 shows how individual docker containers can be started within Java code through the API.

```
DockerTlsManagerFactory factory = new DockerTlsManagerFactory();
TlsInstance tlsServer = factory.getTlsServer(TlsImplementationType.OPENSSL, "1.1.1f");
tlsServer.start();
```

Listing 5: Example of how to start an OpenSSL server with the TLS-Docker-Library.

[5] https://github.com/tls-attacker/TLS-Attacker-Project-Template
[6] https://github.com/tls-attacker/TLS-Docker-Library

Implementations	#Available Versions
BearSSL	3
BoringSSL	21
Botan	48
BouncyCastle	9
DamnVulnerableOpenSSL	1
Firefox	22
GnuTLS	82
JSSE	7
LibreSSL	81
MatrixSSL	18
mbedTLS	127
NSS	34
ocamlTLS	1
OpenSSL	155
PyOpenSSL	11
python-GnuTLS	1
python-mbedTLS	1
rusttls	12
s2n	26
tlslite-ng	74
wolfSSL	84
wolfSSL-py	3

Table 7.2.: Overview of available TLS implementations in the TLS-Docker-Library. Most versions are available as both client and server implementations.

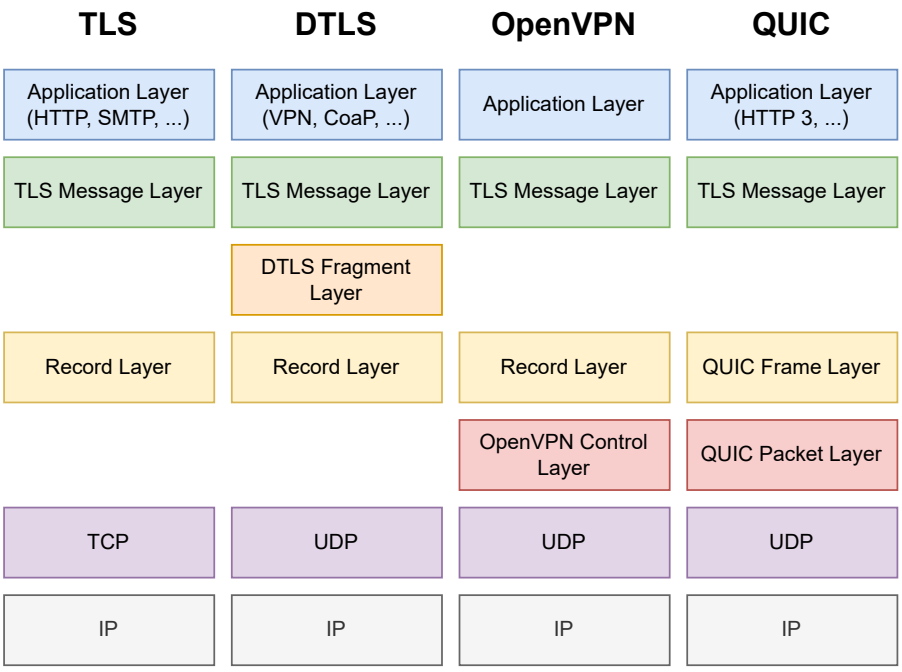


Figure 7.3.: TLS-Attacker 5.0 layer configurations side-by-side. Related protocols still share a lot of common layers, which motivated the layer system and the integration of related protocols into TLS-Attacker.

7.11. Layer System

TLS-Attacker wants to support all kinds of TLS variations, dialects, and configurations. This includes but is not limited to DTLS, QUIC, OpenVPN, and EAP. Implementing these variations became increasingly challenging as they added complexity to the project. To reduce complexity creep, TLS-Attacker will, starting with TLS-Attacker 5.0, implement a layer system that enables a user to define, add, and remove processing layers to the workflow. Each layer does not make any assumption on its position in the layer stack, or about the presence of other layers. Per default, TLS-Attacker supports different layer configurations (visualized in Figure 7.3). However, these configurations can also be changed by the user to create arbitrary configurations. This isolates each protocol layer such that the framework can be easily extended with additional protocol deviates.

8 Scanning the Ecosystem

It is essential for decision-makers and the security community to know how the protocol is used in the real world, as it indicates what kind of features are actually needed in the real world and which are entirely useless. One example of such a feature is the ability of the TLS to define arbitrary elliptic curves for the key exchange. The feature was specified in RFC 4492 [140], but to the best of my knowledge, it was never implemented in any TLS implementation. The same is true for potential issues in the protocol. For example, TLS in CBC mode can be vulnerable to padding oracle attacks ([19,80–83]). Padding oracles and their mitigation are academically well understood. However, it is unclear if these potential issues still threaten real-world deployments without actually looking into real-world deployments. The TLS protocol does not offer solutions to easily extract the supported features or the name of the used software. To extract this information, special security scanners are used that strategically connect to the target system to make conclusions about supported features, the configuration, and present bugs or vulnerabilities. There exist countless tools which can analyze the properties of a TLS server implementation [141–144, 144, 145]. Many TLS scanners are not very sophisticated and focus on common configuration options like the supported cipher suites or the used certificate. This information is usually easy to extract, as it does not require the implementation of cryptography and does not require a deep understanding of the protocol. Tools that can extract more advanced information that test for vulnerabilities within the implemented cryptography are less common. It is not uncommon that the original author of a vulnerability or issue also provided a tool to test for the vulnerability, which then gets integrated into the scanners. For client scanning, a lot less tools are available [146–150]. For TLS clients, a lot of scanners simply parse the `ClientHello` message and present its content. More sophisticated scanners can also scan for support DH group sizes or how the client reacts to different invalid certificates. During my dissertation, I also built a TLS-Scanner, which is based on TLS-Attacker. This allowed me to develop tests that go far beyond simple tests and can enabled me to quickly develop tests for cryptographic components. TLS-Scanner contains code for server as well as client scans. I originally developed the project as part of the SIWECOS [151] project

and was further extended with the help of my colleagues and students that I supervised during their Bachelor's and Master's theses.

8.1. TLS-Scanner

TLS-Scanner is a publically available TLS scanner based on TLS-Attacker[1]. It is designed to allow integration into a web service and to be used by average IT personnel and has the following goals:

1. **Fast.** The scanner should be fast. If a scan would take too long, the user would not frequently use the tool.

2. **Correct.** The scanner should not report false positives. While an expert can look at the results and decide on how to act, showing incorrect results to an average user can spread panic and waste time investigating the issue. This is especially important since some implementations contain bugs that can make scanning difficult, as scanners typically rely on 'somewhat correct' implementations.

3. **Detailed.** The scanner should be as detailed as possible. The more problems the scanner can check for, the better.

4. **Scalable.** We also wanted to use the scanner for Internet-wide scans as part of research projects. For this purpose, it is important that the scanner can scale horizontally.

5. **(Relatively) Safe.** Security scanning is not always risk-free as it sometimes also involves sending invalid messages. Any scanned target may fail to process the input and crash or exhaust the systems' resources. These side effects are to be avoided as much as possible.

6. **Useable.** The audience of the scanner is TLS experts and non-experts (like admins or pentesters). The scanner should therefore be able to present all the details if requested but should present them with indicators, such that non-experts know what to look out for.

7. **Extendable.** It should be easy to add new tests to the scanner.

8.2. Architecture

TLS-Scanner encapsulates each 'test' that is performed within a `Probe`. A `Probe` is a collection of individual, semantically related tests that can be executed together at a specific point within the scan. Once they are started, they do not depend on the results of other `Probes`. Each `Probe` defines a list of requirements that need to be fulfilled before the `Probe` can be executed. These requirements can limit in which cases it makes sense to execute the `Probe` and which information needs to be present for the `Probe` to start. For example,

[1] `https://github.com/tls-attacker/TLS-Scanner`

the `PaddingOracleProbe` requires that the scanner knows which cipher suites and protocol versions are supported by the peer, such that the scanner knows for which cipher suites it should perform the padding oracle scan or if the scan should be omitted entirely (for example if the peer does not support CBC cipher suites). Once executed, the `Probe` stores the results of the `Probe` within a `ScanReport`.

Probe Scheduling. Once the scanning starts, a *scheduler* decides which `Probes` can be executed at any given time. For this purpose, it constantly checks for each `Probe` if its requirements are fulfilled. If they are, the `Probe` is scheduled for execution in a thread pool, which can then execute the `Probes` in parallel. Whenever a `Probe` finished execution (and therefore added new information to the `ScanReport`) each still available `Probe` is reevaluated if its requirements are met. When all running `Probes` are executed, and non can be scheduled anymore, the active scanning portion of the scan is finished.

Probe Execution. Each `Probe` implements an `execute()` function that is executed sequentially by the scanner. This sequential execution is a bottleneck for the scanner if the scanner has a lot of threads. The `Probes` have a different runtime which results in some `Probes` requiring only a single connection to retrieve the desired information, while other `Probes` might require a few hundred. To mitigate this issue, TLS-Scanner has a second thread pool that is solely dedicated to the execution of connections. Whenever a `Probe` wants to perform a TLS connection, it can schedule it for execution. `Probes` can also schedule multiple connections simultaneously, allowing the sequentially executed `Probe` a parallel execution. A `Probe` does not necessarily parallelize all its connections, but whenever a bunch of independent connections is supposed to be executed at the same time, the `Probe` benefits from this parallelization.

Passive monitoring. TLS-Scanner passively monitors all the connections it makes. This allows the scanner to deduce information about the target without performing a new connection. For this purpose, so-called `StatExtractors` look at each connection and extract relevant information from the connection for future analysis.

After probes. `AfterProbes` are special `Probes` that are executed after all regular `Probes` are executed. These `Probes` are not supposed to perform additional handshakes, but to reason about the already collected information. This can also involve information from passive monitoring. An example for an `AfterProbe` which does not require additional handshakes and uses information from the passive monitoring is the `DhValueAfterProbe`. The `Probe` looks at all the Diffie-Hellman public keys the scanner has seen during the regular scans and then makes decisions about the quality of the keys (safe primes, key reuse, composite moduli). Extracting this information passively is more reliable and efficient than active scans, as some servers use different sets of Diffie-Hellman parameters due to CDN setups.

Rating. After all `AfterProbes` are executed, all facts about the target are extracted. In the next step, the scanner attaches meta-information about the `ScanReport` which judges the quality of the scanned server and its configuration. This quality assessment is very subjective as the risks involved from certain vulnerabilities or misconfigurations can be rated differently. Our developed rating system is based on points. Whenever a server has a beneficial property it gets points added to its score, while unwanted properties reduce points. To avoid the situation where a server with an otherwise good configuration but a critical vulnerability, for example, Heartbleed [152], gets a lot of points, some properties limit the number of points the server can achieve. After the score is computed, the rater outputs an explanation of how the score was computed and advice on how to improve the score. The default rating system within TLS-Scanner was built in cooperation with Juraj Somorovsky as part of the Future Trust [2] project. The rating system is easily interchangeable with XML files.

Compliance. At last, the scanner evaluates to what extent the server complies with certain guidelines. At the time of writing the scanner can evaluate compliance towards NIST SP 800-52r2 [3] and BSI-TR-02102-2.[4]

8.2.1. Supported Features

The Scanner supports client and server scanning, while client scanning is still a fairly new feature in the scanner it was not developed by me. It is therefore omitted in this dissertation. At the time of writing the TLS-Server-Scanner supports 42 `Probes` and 12 `AfterProbes`. An overview of the implemented probes is given in Table 8.1, while an overview of all `AfterProbes` is given in Table 8.2.

8.2.2. Side-Channel Analysis

The analysis of side channels is a common task for TLS-Scanners. To evaluate them, TLS-Scanner needs to send inputs to the target, called *test vectors*, and has to observe how the server reacts to the test vector and look for potential side channels. Automatically analyzing side-channel vulnerabilities can be error-prone when doing black box tests. To discover a side channel the scanner has to compare the responses from the peer and find a measurable difference. Since TLS-Scanner is supposed to work as a remote scanner, the side-channel that can be considered is limited (for example, power consumption is not directly measurable remotely). In principle, TLS-Scanner can consider the transmitted network packets, their contents, and the timings of the packets as potential side channel sources. While the usage of timing information is likely relevant for side channels, it is difficult to evaluate with blackbox tests as timing measurements are noisy. As this directly conflicts with TLS-Scanner's goal to provide reliably

[2] `https://pilots.futuretrust.eu`
[3] `https://doi.org/10.6028/NIST.SP.800-52r2`
[4] `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102-2.html`

Probe	Main Authors	Description
Alpaca	Robert Merget	Checks whether the server has mitigated the Alpaca Attack by checking if the server uses strict ALPN and SNI.
ALPN	Robert Merget	Evaluates which ALPN protocols the server is willing to negotiate.
Bleichenbacher	Juraj Somorovsky Robert Merget	Evaluates if the server is vulnerable to a Bleichenbacher attack with direct responses. See Subsection 8.2.2.
CCA	Mario Korth	Evaluates which certificates the server accepts in an mTLS connection. This `Probe` especially also tries to find bypasses.
CCA Required	Mario Korth	Evaluates whether the server also accepts clients that do not provide a certificate or whether client certificates are enforced.
CCA Support	Mario Korth	Evaluates whether the server requests client authentication.
Certificate	Robert Merget Marcel Maehren	Performs multiple handshakes in different configurations to retrieve all available server certificates.
Cert. Transparency	Jannis Pinter	Retrieves SCTs and their contents.
Cipher Suite Order	Robert Merget	Checks whether the server negotiates cipher suites according to its own personal order.
Cipher Suite	Robert Merget	Evaluates which cipher suites the server supports in which protocol version.
Common Bugs	Robert Merget	Checks if the server suffers from common bugs or intolerances.
Compression Algorithms	Robert Merget	Evaluates which compression algorithms the server supports.
Direct Raccoon	Robert Merget	Evaluates whether the server is vulnerable to the direct variant of the Raccoon attack.
DROWN	Nimrod Aviram	Evaluates whether the server is vulnerable to the Drown attack, and if so, which variants.
Common DTLS Bugs	Nurullah Erinola	Checks if the server suffers from common DTLS-specific bugs.
DTLS Features	Nurullah Erinola	Tests support for DTLS-specific features (i.e., fragmentation).
DTLS HVR	Nurullah Erinola	Tests if a server uses DTLS cookies and which values the server uses for its DTLS cookie computation.
DTLS Message SQN	Nurullah Erinola	Evaluates the peer's behavior in regards to message sequence numbers.
DTLS Retransmissions	Nurullah Erinola	Evaluates the behavior of the server in regards to retransmissions.
Early CCS	Juraj Somorovsky	Evaluates whether the server is vulnerable to the EarlyCCS attack.
EC Point Format	Marcel Maehren	Evaluates which EC point formats the server supports.
ESNI	David Ziemann	Evaluates whether the server supports ESNI (draft-2).
Extension	Robert Merget	Evaluates which TLS extensions the server supports/reflects.
Heartbleed	Juraj Somorovsky Robert Merget	Checks if the server is vulnerable to the heartbleed vulnerability.
Hello Retry Request	Marcel Maehren	Tests if the server sends TLS 1.3 `HelloRetryRequest` messages.
HTTP False Start	Robert Engel	Tests if the server accepts HTTP false start.
HTTP Header	Robert Merget	Tests which HTTP headers the server sends.
Invalid Curve	Juraj Somorovsky Marcel Maehren	Checks if the server is vulnerable to invalid curve attacks.
MAC	Robert Merget	Checks if the server is verifies MAC's and the `verify_data` in the `Finished` message. The `Probe` is deactivated by default.
Named Curves Order	Bastian Haverkamp	Checks whether the server named groups according to its own personal order or whether the server uses the client preferred order.
Named Groups	Robert Merget	Evaluates which named groups the peer supports.
OCSP	Nils Hanke	Evaluates the OCSP server that is specified in the certificates by sending OCSP requests to it.
Padding Oracle	Robert Merget	Evaluates if the server is vulnerable to a padding oracle attack with direct responses. See Subsection 8.2.2.
Protocol Version	Robert Merget	Tests which protocol versions the peer supports.
Record Fragmentation	Phillip Nieting	Tests if the peer supports defragmentation of messages.
Renegotiation	Robert Merget	Evaluates if the peer supports client-side renegotiation and whether it is vulnerable to a renegotiation attack.
Resumption	Robert Merget	Tests whether the server supports session resumption.
Session Ticket Zero Key	David Ziemann	Tests if the server is vulnerable to CVE-2020-13777 testing if its session tickets are decryptable with a zero key.
Sig. & Hash Algorithm	Robert Engel	Evaluates which signature and hash algorithms the peer supports.
Sig. & Hash Algorithm Order	Dominik Vorjohann	Checks whether the selected signature and hash algorithms are selected according to the client's preferred order.
SNI	Robert Merget	Tests if the SNI extension is mandatory to speak to the server.
Fallback SCSV	Robert Engel	Tests if the server correctly implements the downgrade protection from RFC 7507.
Token binding	Robert Merget	Evaluates which Token binding parameters are supported and whether Token binding is implemented securely.

Table 8.1.: Overview of implemented TLS `Probes` in the TLS-Server-Scanner.

AfterProbe	Main Authors	Description
Certificate Sig. & Hash Algorithm	Robert Engel	Analyzes the signatures and hash algorithms of gathered certificates.
Destination Port	Nurullah Erinola	Analyzes if the server switches the source port in its response to incoming connections.
Diffie-Hellman Values	Robert Merget	Analyzes seen DH public keys in regards to their strength and whether the server is repeating keys or not.
DTLS Retransmission	Nurullah Erinola	Analyzes the retransmission capabilities of a DTLS server.
EC Public Key	Robert Merget	Analyzes seen EC public keys on whether the server is repeating keys or not.
FREAK	Robert Merget	Analyzes if the server is vulnerable to the FREAK attack.
Logjam	Robert Merget	Analyzes if the server is vulnerable to the Logjam attack.
PO Identification	Robert Merget	Analyzes padding oracle vulnerabilities and tries to identify them.
POODLE	Robert Merget	Analyzes if the server is vulnerable to the POODLE attack.
Raccoon	Robert Merget	Analyzes if the server is vulnerable to the Raccoon attack.
Randomness	Bastian Haverkamp	Analyzes seen random values, like the server random or CBC IVs, and performs statistical tests on them.
Sweet32	Robert Merget	Analyzes if the server is vulnerable to the Sweet32 attack.

Table 8.2.: Overview of implemented TLS `AfterProbes` in TLS-Server-Scanner.

correct results, TLS-Scanner does not consider timing information for the analysis of side channels. Instead, TLS-Scanner solely relies on the contents of the received network packets. In particular, TLS-Scanner only considers the TCP contents and wether the TCP stream was closed with a TCP-FIN or TCP-RST. This may overlook some side channels; for example, side channels based on the TCP fragmentation of the data stream are not detected. Considering these side channels would require using of lower level OS APIs than typically possible from within a Java application, which would require a more sophisticated networking architecture. It was decided to ignore such side channels in favor of a simpler architecture. Interestingly, Drees et al. [153] also considered side channels based on other lower level fields on the TCP layer (but also excluded timing information). They used machine learning to find side channels in the gathered data. It is generally beneficial to consider more available data points, however, there is no public documented case where considering these additional fields resulted in a real-world side-channel vulnerability.

Statistical Tests. Considering the above-mentioned side-channel sources, it can still be tricky to determine if a real side-channel vulnerability is present, even if a difference is observed. Consider the following example: During a Bleichenbacher test, the scanner sends two test vectors, A and B. Vector A contains a PKCS#1.5 `ClientKeyExchange`, a `ChangeCipherSpec`, and an invalid encrypted `Finished` message, while vector B contains the same messages, but the `ClientKeyExchange` message contained an invalid PKCS#1.5 padding in the first two bytes. If an attacker can observe a measurable difference between the processing of these two vectors, the peer is considered vulnerable to a Bleichenbacher attack [12]. But now consider that the scanner received a `BAD_RECORD_MAC` and a `TCP-FIN` in response vector A while it received a `BAD_RECORD_MAC` from the server (without closing the socket). Naiivly we would believe that given this information, we could conclude that the server is vulnerable to a Bleichenbacher attack, as we can differentiate the two vectors based on the state of the TCP socket, which in return is observable by the attacker. The problem with this assumption is that it assumes that servers react deterministically towards our vectors, which is not necessarily the case in real-world

deployments. For example, it could be that the server was replying with `TCP-FIN` for vector B, but the scanner did not receive that packet yet. This could be due to timeout, lost packets, or network equipment failure between the scanner and the server. It is also possible that the server did really not close the TCP connection, but is still not vulnerable because the event is completely unrelated to the content of the decrypted `ClientKeyExchange` message.

Another common source of non-determinism seen in the real world are heterogenous CDN deployments. In CDN deployments, there is not a single server the scanner is scanning, but the scanner is scanning multiple different servers at the same time, as the scanner cannot choose the exact server it is talking to. This is typically not problematic if all servers are on the same patch level and run the same software on similar hardware. In real-world deployments, this is not necessarily the case. This can result in entirely different responses to the vectors of the scanner. For example, in response to vector A, the scanner could receive a `BAD_RECORD_MAC` alert, while in response to vector B, the scanner could receive a `RECORD_OVERFLOW` alert. This could simply be the result of two different server implementations, which are both in it for itself secure, but the combination appears towards the scanner as non-deterministic and could therefore be misclassified as a false positive.

To truly show that a potentially non-deterministic server is vulnerable to a side channel attack, we have to show that the behavior difference we observe is related to the difference in the processing of our vectors. If the response is unrelated to our vectors, we would expect that the responses we receive are evenly distributed among our vectors during multiple executions. If the distribution is uneven, the vector choice influences the responses from the server, resulting in a related measurable difference for the attacker. We can use a statistical test to test if the responses we receive are evenly distributed among our vectors. TLS-Scanner uses Fisher's exact test [154] if there are only two different responses and the χ^2 [155] test if there are more than two responses. Both tests produce a p-value representing a confidence value indicating how certain the test is that the seen response distribution is evenly distributed. Typically, p-values lower than 0.01 to 0.05 are desired. For our purpose, these values are still too high, as the scanner does a lot of side-channel tests which would result in an unacceptable amount of false-positive classifications. The scanner, therefore, uses a much smaller p-value threshold to compensate for this (currently set to 0.001). In the real world, the observed p-values are typically much smaller (10^{-8}).

Timeout. Some servers never respond to the messages of the scanner. If the scanner does not receive an answer, it typically does not know if the server has not yet responded or if it will never respond. Additionally, the server's answers may span multiple TCP packets, so there is no reliable way to ascertain whether the scanner has received the server's answers in full at any point in time. The scanner, therefore, has to implement a timeout. It is critical to select an appropriate timeout as if the timeout is too low, the scanner might miss responses due to high server load. Conversely, a high timeout value would decrease the scanning performance. We empirically determined that a timeout

of one second works well in practice, and mostly guarantees that the server did have enough time to process our record and respond. However, even when using this timeout value, we found servers that responded non-deterministically due to high loads or various bugs.

8.3. Probes Details

The TLS-Scanner project implements a huge amount of `Probes` to retrieve all sorts of information. Explaining each and every `Probe` in detail is outside the scope of this dissertation. Still, some of the `Probes` and their concepts are worth highlighting, especially since they are relevant for other sections in this dissertation. A list of all implemented `Probes` can be seen in Table 8.1, while Table 8.2 presents all `AfterProbes`.

8.3.1. Supported Feature Discovery

A common pattern for `Probes` that is seen throughout the scanner is the discovery of features. In TLS, the client sends a list of its supported features in its `ClientHello` message, while the server simply chooses features from the list it also supports. A naive approach to discover if a given feature is supported by the server is to simply send a `ClientHello` which contains said feature, with all over features of the same kind disabled. This technique leaves the server no choice but to choose the offered feature. If a choice is made, the feature is supported, while if the connection is aborted, the feature is not supported. The problem with this approach is that it requires an individual connection for each individual feature the scanner is testing for. For example, when scanning for supported cipher suites, this approach would require an individual connection for each *potentially* supported cipher suite, as each cipher suite would require its own connection. With hundreds of officially standardized cipher suites, the overhead of this technique becomes apparent quickly.

A better approach is to first advertise support for all supported features and let the server pick one. In a subsequent connection, the scanner advertises all supported features but removes the ones the server has already chosen from the list. Once the server no longer chooses a feature (and terminates the connection early), we know that exhausted we all supported values. This way, we only require one connection for each supported feature (+1 for the final connection), while the naive approach requires one connection for each potentially supported feature, resulting in huge performance improvements.

8.3.2. Padding Oracle Probe

The `PaddingOracleProbe` is designed to identify non-timing based padding oracle vulnerabilities in server implementations. Previous vulnerabilities within TLS have shown that the vulnerabilities may appear only in very specific components in the system, while seemingly related components are not affected. Therefore, to ensure that a server does not exhibit any padding oracles at all, it

is important that the `PaddingOracleProbes` touches as many related security components as possible.

The `PaddingOracleProbe` depends on the cipher suite and the `ProtocolVersionProbe`. The `Probe` excludes all export and anonymous cipher suites from these tests since they are already trivially broken by a MitM attacker. It then performs an individual scan for each CBC cipher suite and its supported protocol version.

Test Vector Generation. To detect padding oracles in implementations, we first connect to the target and then send a malformed record. Each malformed record is invalid in regards to either the padding, MAC, and application data or a combination of these. We then observe the responses with the technique described in Subsection 8.2.2. It is infeasible to test each possible malformed record, as there are roughly $2^{(2^{14*8})}$ different vectors that the scanner could potentially send.[5] It is therefore only possible to test a subset of all possible vectors. An ideal scanner would send as few vectors as possible while sending as many vectors as necessary to detect every possible vulnerability. Vulnerabilities are *potentially* arbitrary. For example, a vulnerable implementation could correctly check all padding bytes unless the padding bytes are exactly 16 bytes long. In this case, the implementation does not check a specific bit in the padding.[6] We therefore carefully selected a set of malformed records which are motivated by previous research, for which we believed that they are likely to trigger different behavior and therefore indicate a padding oracle vulnerability.

Admittedly this way of selecting the set of malformed records means we can only detect vulnerabilities that are similar to known ones. However, this approach is cost-effective and well-suited even for large-scale scans. Since only a limited number of messages can be sent to individual servers during large-scale scans, automatic approaches for the test vector generation are infeasible.

During our test, it is important that any difference in behavior we observe is directly related to the cryptographic operation and that the records cannot be distinguished on the ciphertext alone. One important property, therefore, is that every record in a given test is of equal length. Additionally, it is important that the length of the record is big enough such that the record cannot be discarded by the implementation before the record is decrypted (to avoid false negatives); for example, recent OpenSSL versions respond with an error message if the encrypted TLS record is shorter than the MAC length.

The `PaddingOracleProbe` uses malformed records that are all 80 bytes in length. We decided to use 80 bytes to have enough room for an HMAC output combined with two full padding blocks independent of the used HMAC algorithm, which in return also allowed us to test optional non-minimum-length paddings. In total we designed 25 malformed records, Table 8.3 shows a sum-

[5] This is the number of all possible vectors. This includes the number of valid vectors unsuitable for padding oracle tests. But since most possible vectors are suitable, it gives a well enough approximation.

[6] The above behavior may sound contrived, but similar behaviors have been found in the wild, see e.g. [15, 82, 83].

Nr.	MAC			Padding		
	Len	Pos	Modification	Len	Pos	Modification
1	20	20	⊕ 0x01	56	–	–
2	20	11	⊕ 0x08	56	–	–
3	20	1	⊕ 0x80	56	–	–
4	19	1	DEL	56	–	–
5	19	20	DEL	56	–	–
6	0	–	–	80	ALL	0x4F
7	0	–	–	80	ALL	0xFF
8	20	–	–	60	1	⊕ 0x80
9	20	–	–	60	31	⊕ 0x08
10	20	–	–	60	60	⊕ 0x01
11	20	1	⊕ 0x80	60	–	–
12	20	9	⊕ 0x08	60	–	–
13	20	16	⊕ 0x01	60	–	–
14	20	1	⊕ 0x01	60	1	⊕ 0x80
15	20	1	⊕ 0x01	60	31	⊕ 0x08
16	20	1	⊕ 0x01	60	60	⊕ 0x01
17	20	–	–	6	1	⊕ 0x80
18	20	–	–	6	3	⊕ 0x08
19	20	–	–	6	6	⊕ 0x01
20	20	1	⊕ 0x80	6	–	–
21	20	9	⊕ 0x08	6	–	–
22	20	16	⊕ 0x01	6	–	–
23	20	1	⊕ 0x01	6	1	⊕ 0x80
24	20	1	⊕ 0x01	6	3	⊕ 0x08
25	20	1	⊕ 0x01	6	6	⊕ 0x01

Table 8.3.: A summary of our malformed records, as constructed for `TLS_RSA_WITH_AES_128_CBC_SHA`. The columns indicate length, position, and modification for MAC and padding bytes, respectively. ⊕ denotes XOR'ing the listed value in the listed position. DEL denotes deleting one byte in the listed position.

mary of these malformed records for the case of `TLS_RSA_WITH_AES_128_CBC_SHA`.

Flipped MAC bits. We start with a valid record containing application data, a MAC, and four padding bytes. We then create three malformed records based on this record: One by flipping the most significant bit in the first MAC byte, one by flipping a middle bit in the middle of the MAC bytes, and one by flipping the least significant bit of the last MAC byte. We chose these malformed records to detect implementations where the MAC is not completely checked. The specific bit flipping positions are motivated by the recent OpenSSL vulnerability [156], where OpenSSL only checked the least significant bit of each byte on some platforms, and by further vulnerabilities caused by incomplete MAC validations [82, 83].

Missing One MAC byte. We start with a valid record containing empty application data, but with valid MAC and padding. We then modify it to create

two malformed records: One where we delete the first MAC byte, and one where we delete the last MAC byte. We then add another padding byte in both messages. These malformed records could also trigger vulnerabilities caused by incomplete MAC validations and are indirectly motivated by research from Yngve Petterssen [83].

Missing MAC. Motivated by CVE-2016-2107 [87], we created two malformed records which only contain padding and do not contain a MAC at all: One where we supply exactly 80 bytes of valid padding (`0x4F`), and one where we supply 80 bytes of incomplete padding of value `0xFF`. The latter is not only missing the MAC but also contains invalid padding since if the value of the last byte is `0xFF`, there should be 256 padding bytes.

Combining Valid and Invalid MAC and Padding. The last group of malformed records is messages containing valid and invalid MAC and padding of three types: valid MAC and invalid padding, invalid MAC and invalid padding, and invalid MAC and valid padding. We create three sub-types for each of these three types, depending on which bit positions we flip; we flip either the most significant, middle, or least significant bit in the first, middle, or 16th byte, respectively. For each of these nine sub-types, we create one version which contains application data, and one without. The length of the application data is chosen such that the padding bytes are contained within one plaintext block, while the malformed records without application data contain more than one block of padding. This aims to detect implementations that check only the last block of padding bytes.

8.4. TLS-Crawler

The previously mentioned TLS-Server-Scanner is specifically designed to test a single host. For academic purposes, it is often desirable to scan not only a single server, but a whole range of servers (or even a whole address space) to get a good picture of the state of the TLS ecosystem.

To perform large-scale studies with TLS-Scanner, we developed TLS-Crawler. TLS-Crawler spreads a lot of individual TLS-Scanner instances across multiple machines to still be able to scan larger amounts of hosts. For this purpose TLS-Crawler uses a MongoDB[7] to store the results, and RabbitMQ[8] as a synchronization mechanism. TLS-Crawler itself consists of two components, a controller and a worker. To start a scan with the crawler, the controller has to be started, which publishes information about the targets and the nature of the scan in RabbitMQ. On the other hand, the workers can be spread across multiple machines to improve the scanning speed by distributing the overall workload. Each worker looks into RabbitMQ to see which scans are currently active, and then grabs several hosts from RabbitMQ and schedules them locally for scanning. After these hosts are scanned, the crawler pushes the results into

[7] `https://www.mongodb.com`
[8] `https://www.rabbitmq.com/`

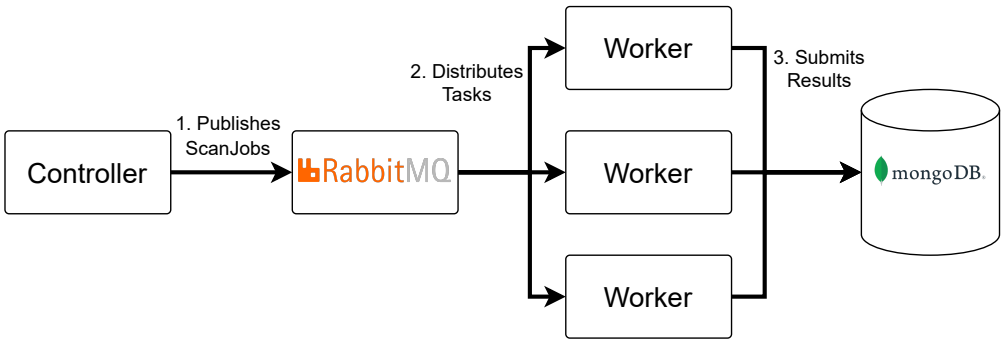


Figure 8.1.: Our TLS scanning infrastructure is based on mongoDB a well-established components for data persistence and RabbitMQ, a well known message broker. Workers use multiple instances of TLS-Scanner, which internally uses TLS-Attacker to extract the relevant information.

MongoDB and notifies RabbitMQ that the tasks were successfully executed. It then starts over by looking into RabbitMQ for new tasks. If a worker crashes or gets turned off, a timeout within RabbitMQ will run out which adds the tasks back into the task queue, such that other workers can process the tasks. All componentens within the crawler allow for horizontal scaling. If one component is too slow for the current task, it can be speed up by simply deploying more instances of it, for example, more workers, more MongoDB instances or more RabbitMQ instances. The TLS-Scanner instances on a single worker share the same threadpool for the actual execution of the handshake connections. The shared thread pool and the parallelization of the TLS-Scanner instances effectivly shuffles the hosts in the scheduling. This shuffling effect reduces the load on each individual scanned host as the handshakes are spread across a longer timeframe. Figure 8.1 visualizes the TLS-Crawler architecture.

Comparision to ZGrab. Many tools like ZGrab [157] or Massscan [158] already existed before TLS-Crawler that were able to scan the TLS ecosystem as well. Especially ZGrab is widely popular in the scientific research community and has been used in multiple publications [157]. ZGrab is a blazing fast network scanner built to scan application layer protocols for various properties. The key to ZGrab's speed is a precomputed payload and a reduction of local state. By omitting as much local state as possible, the scanner can initially send many packets and then react to incoming packets without having a single thread or process for every scanned host. In the context of TLS, the approach can answer many research questions, especially in regard to the certificates used on the scanned hosts. However, the approach lacks the possibility to perform scans that require more (or even a lot) of state to be successful. For example, the scanning technique presented in Subsection 8.3.1 is not well suited for ZGrab, as the scanner would need to keep track of supported cipher suites of each server to compute the following `ClientHello` message. With the traditional ZGrab approach, scanning for all supported cipher suites would require

an individual ZGrab scan for each supported cipher suite. In comparison to ZGrab, TLS-Crawler is slow. TLS-Crawler is typically unable to make use of all the available network bandwidth and is limited by the CPU and the available RAM. The strength of TLS-Crawler lies in its ability to use TLS-Scanner, which in turn can use TLS-Attacker, which gives the framework a very flexible framework for rapid prototyping. It is easy, fast, and cheap to develop new `Probes` and to quickly get insights into the TLS ecosystem, independent of the state requirements of the `Probe`. The number of servers that can be scanned with TLS-Crawler therefore highly depends on the availability of CPU and RAM. In our internal experiments, we typically analyzed 100.000 to 1.000.000 servers. ZGrab is usually used on the IPv4 adress space, containing roughly 60 million hosts on port 443. Many experiments do not require a complete view of the ecosystem, but for them it is enough to select a sample of the whole population to estimate the overall population. This sample typically involves a random selection or a some the most popular targets, as these produce the most relevant results. For this purpose, the sample sizes achievable with TLS-Crawler (with modest hardware) are usually enough to get an accurate picture of the entire ecosystem.

9 Large-Scale Analyses of TLS Attacks

For the TLS community, it is very important to have a good overview of the deployed implementations and their configurations. The protocol leaves many choices to the developers in regards to which aspects and features of the protocol get implemented. Most of the time, it is important that both, the client and the server support a given feature. It is therefore very important for developers to know if it is even 'worth it' to implement a feature. Additionally, it is important for the community to know to which extent potential protocol issues like padding oracle vulnerabilities really manifest in real-world deployments to decide if changes in the standard are required to improve the situation. During my dissertation, I conducted multiple studies of the TLS ecosystem in regard to many different aspects. This section introduces three of them, a study on the Raccoon attack (Section 9.1), a study on the Alpaca attack (Section 9.2), and finally an in-depth evaluation of CBC padding oracles (Section 9.3).

9.1. Evaluation of the Raccoon Attack in the TLS Ecosystem

Recall from Chapter 5, that the Raccoon attack is a side-channel attack that attacks connections that use FFDHE with a static or semi-static public key and tries to compute the PMS of the connection.

9.1.1. Methodology

To estimate the impact of the Raccoon attack on currently deployed servers, we conducted a scan among the Alexa Top 100k on port 443. We evaluated how common static-DH cipher suites are by trying to negotiate them. We also evaluated how prevalent key reuse is in TLS-DHE and analyzed the used modulus sizes. We also tried to find servers that are vulnerable to a direct oracle by sending `ClientKeyExchange` messages, which either resulted in a PMS with a leading zero byte or not, and by observing the server's behavior. For this purpose, we used techniques from Subsection 8.2.2 and carefully observed the TCP connection state, also with a shortened message flow. We performed each

handshake three times to rule out inconsistencies in the server behavior. If a server showed at least occasionally different behavior, we performed additional handshakes (97 each) to collect more data on the issue for a total of 100 connections.

9.1.2. Results

Our scans were conducted in April 2020.

DH & DHE support. The results of our scan are shown in Table 9.1. In total, 86607 servers of the scanned servers supported SSL/TLS. A total of 32% of the scanned servers supported DHE cipher suites. Only a single server advertised support for static DH cipher suites.

Key Reuse. Although servers typically reuse ephemeral keys until they are restarted, it is not enough to monitor the ephemeral key in two consecutive handshakes to conclude if a server reuses ephemeral keys, as many hosts are using load balancing setups in which multiple different TLS servers are handling incoming connections. Usually, each server manages its own ephemeral keys, and these keys are not shared across servers. Since we do not know how many potential servers are within a load balancing setup, we do not know how many handshakes we have to perform to make sure that we can detect key reuse. To overcome this issue, we observed all public keys transmitted to us during the scanning process. If we observed at least one reuse, we considered the server as reusing ephemeral keys. We did not evaluate how long these servers reused their public keys, as this would require a longitudinal study outside the scope of this work.

Our scans showed that a total of 3.33% of the scanned servers reused their ephemeral DH keys. This is slightly lower than the 4.4% reported by Springall et al. in 2016 [106], but note that Springall et al. scanned the Alexa Top 1 Million, whereas we scanned the Alexa Top 100K. Furthermore, DHE support by major clients declined during this period, and OpenSSL removed ephemeral key reuse in 2016. Springall et al. did measure the time period for key reuse, and found that 1.3% of servers supporting DHE reused values for at least one day, and 1.2% for at least 7 days.

Key lengths. The key lengths used by the scanned servers can be seen in Table 9.1. The data shows that servers that reuse ephemeral keys generally tend to use weaker keys than servers that do not.

Perfect direct oracles. A total of 87 servers were exhibiting a perfect direct oracle as described in Section 5.6, meaning that they were reliably showing different behavior based on the leading zero byte of the PMS. Almost all of these servers (84) were reusing their ephemeral keys. We fingerprinted this vulnerability and were able to attribute most of the discovered oracles to devices of the company F5. F5 confirmed the vulnerability and released a patch on

Modulus sizes	# Domains	# With key reuse
1024 bits	5310	2213
2048 bits	23428	1116
4096 bits	3045	4
8192 bits	1	0
Other	277	0
Total	32061	3333

Table 9.1.: The observed key lengths for DHE cipher suites in the Alexa Top 100k scan. Units denote the number of domains with the corresponding key length.

the 9th of September 2020 in Security Advisory K91158923 (CVE-2020-5929). These servers were sending either one or two handshake failure alerts depending on whether the PMS started with a zero byte or not. The vulnerability was not present on all supported cipher suites. Note that an attacker could still use any vulnerable cipher suite to attack the connection of a non-vulnerable cipher suite.

Imperfect Direct Oracles. We found 815 servers that did not show a perfect oracle, meaning that they did not allow for a distinction with every executed handshake, but only occasionally showed a distinguishing behavior. We assume that we observe this behavior because of another factor that we did not control (or cannot control) that influences the behavior difference. These factors may include CDN setups, where only parts of the CDN are vulnerable, internal memory allocations, network issues, or resource shortages. We did not exclude these hosts from our study but investigated if the behavior difference correlates with a leading zero byte in the PMS or not. Any behavior difference unrelated to a leading zero byte is expected to happen with roughly the same probability on all executed handshakes. If the difference is somehow related to a leading zero byte, we should see a non-uniform distribution of the responses, which can be used by an attacker to distinguish if the PMS for a given `ClientKeyExchange` message will start with a zero byte or not. We used the technique from Subsection 8.2.2 to detect side channel related to a leading 0 byte in the PMS. For each host, we tested each cipher suite in each protocol version individually and accepted all hosts as vulnerable for which the p-value on one of the executed tests was smaller than 10^{-9}. Given these tests, we discovered that a total of 815 servers (excluding perfect oracles) showed an observable difference based on a leading zero byte in the PMS. However, none of these servers were reusing ephemeral public keys. As of the time of writing, we do not know which implementation is responsible for this behavior.

9.2. Evaluation of the Alpaca Attack in the TLS Ecosystem

The Alpaca attack, which was presented in Chapter 6 is a cross-protocol attacks that exploits a flaw in the authentication of TLS.

9.2.1. Methodology

To estimate the impact of the Alpaca attack we evaluated the number of HTTPS servers that are vulnerable to cross-protocol attacks with SMTP, IMAP, POP3, or FTP in an Internet-wide scan of the IPv4 address space. Choosing all protocols is computationally infeasible for us. For each server, we looked for servers with trusted and compatible certificates. Additionally, we analyzed how these servers react to invalid server names with SNI and how they react if they cannot choose a valid application layer protocol with ALPN.

To evaluate how many application servers have trusted certificates compatible with HTTPS servers, we conducted multiple IPv4 scans on standard and well-known application ports for SMTP (25, 587, 465, 26, 2525), IMAP (143, 993), POP3 (110, 995), and FTP (21, 990) using ZMap [159] and ZGrab 2.0.[1] We excluded all hosts that could not complete a TLS handshake. We then determined which of these servers have a trust path to a generally trusted root CA. We considered a CA as generally trusted if it is trusted by either Mozilla, Google, Microsoft, Apple, Oracle, or OpenJDK. We then gathered all trusted certificates and extracted their Common Names (CN) and Subject Alternative Names (SAN) to find corresponding HTTPS servers. For entries that contained a *, we guessed the subdomain by replacing * with `www`. We then tried to connect to these hostnames on port 443 using the HTTPS protocol and collected the presented certificates.

We performed two more scans on those SMTP, IMAP, POP3, and FTP application servers that offered a trusted certificate. In the first scan, we estimated the number of application servers that tolerate incorrect SNI hostnames by performing a TLS handshake with the SNI hostname `example.com`. We recorded if the TLS handshake completes successfully despite the mismatching hostname. In the second scan, we estimated the number of application servers that tolerate incorrect application layer protocols by performing a TLS handshake with the same ALPN extension as sent by the Chrome web browser. We recorded if the TLS handshake completes successfully despite the mismatching protocol identifiers.

9.2.2. Results

Our scans were conducted between July and October 2020, the results of our scans can be seen in Table 9.2. Across all protocols, 62,85% of the discovered TLS application servers used generally trusted certificates. A notable outlier is FTP on port 21, where the number of trusted certificates were only 44%. A possible explanation is that FTP server certificates are often signed by private

[1] `https://github.com/zmap/zgrab2`

CAs that are not generally trusted by browsers. We found that about 25% of the untrusted FTP certificates were signed by such private CAs.

TLS Version. Previous studies analyzing the TLS ecosystem have shown that servers running SMTP, IMAP, or FTP do not offer timely TLS protocol support [93,133,160]. Running a service with outdated TLS versions can negatively affect cross-protocol attack execution because current browsers only support TLS 1.2 and TLS 1.3.

Our scan does not include servers supporting only TLS 1.3 due to a lack of support in the version of ZGrab we used, but we suspect that the number of such exclusive servers is marginal among the long-established protocols we analyzed. Our scans show that across all analyzed protocols, 90% to 96% of the scanned application servers with trusted certificates support TLS 1.2, while the rest only support older versions. This means that successful attack exploitation can fail in at most 10% of these servers due to missing support for TLS 1.2.

ALPN and SNI. We removed all host responses from the data set for which the handshake was either successful or could be attributed to an unrelated error (such as connection timeout), and were left with a marginal number of hosts which potentially rejected the TLS handshake because of the ALPN or SNI extension. Depending on the protocol and port, we can give an upper bound for servers potentially supporting ALPN or SNI correctly below 0.5%. We conclude that ALPN or SNI do not pose an obstacle to cross-protocol attacks today.

Web Servers Vulnerable to Cross-Protocol Attacks. Across all analyzed protocols, we collected a total number of 2,088,328 distinct hostnames. Our search for HTTPS servers on those hostnames revealed a total of 1,441,628 HTTPS servers for which at least one SMTP, POP3, IMAP or FTP server exists that was using a generally trusted certificate, which is 69% of all the unique hostnames scanned. Of these web servers, 24,202 were in the Tranco 1M list [161] of the most prominent hosts on the Internet.[2] This means that for the majority of the servers with trusted certificates on SMTP, POP3, IMAP, or FTP, there exists an HTTPS server with a compatible certificate vulnerable to a general TLS cross-protocol attack, where application data is processed by the substitute server rather than the intended web server.

[2]Downloaded on 2020-10-11.

			Server IPs with TLS		Certificate Names (CN & SAN)	
Protocol	**Port**	**STARTTLS**	**Total**	**Valid Certificate**	**# Unique**	**# HTTPS**
SMTP	25	Yes	3,427,465	1,744,052 (50,88%)	1,048,090	782,710 (74.68%)
SMTP	587	Yes	3,495,626	2,471,893 (70,71%)	1,176,374	821,534 (69.85%)
SMTPS	465	-	3,511,544	2,450,062 (69,77%)	1,046,240	724,557 (69.27%)
SMTP	26	Yes	565,672	514,425 (90,94%)	130,624	79,234 (60.66%)
SMTP	2525	Yes	231,009	139,536 (60,40%)	50,514	31,009 (61.40%)
IMAP	143	Yes	3,707,577	2,463,293 (66,44%)	1,103,455	782,410 (70.92%)
IMAPS	993	-	3,919,999	2,597,232 (66,26%)	1,287,370	926,313 (71.97%)
POP3	110	Yes	3,551,226	2,342,545 (65,96%)	983,912	690,111 (70.15%)
POP3S	995	-	3,828,411	2,580,379 (67,40%)	1,170,197	848,744 (72.56%)
FTP	21	Yes	4,826,891	2,130,271 (44,13%)	675,432	421,923 (62.48%)
FTPS	990	-	305,646	282,382 (92,39%)	115,292	95,197 (62.73%)
Total			31,371,066	19,716,070 (62,85%)	2,088,328	1,441,628 (69.03%)

Table 9.2.: Results from our Internet-wide scan by protocol and port (July to October 2020). We first give the number of IP addresses that provide the given service and allow a successful TLS handshake to be made. Then we show the number of those IP addresses that offer a certificate that is considered valid for a browser (except for hostname matching). Next, we give the number of unique names found in the CN and SAN of the valid certificates. Finally, we give the number of HTTPS servers we found among these names, with `*` replaced by `www` as the most common guess for web servers using wildcard certificates.

9.3. Evaluation of CBC-Padding Oracle Attacks in the TLS Ecosystem

The CBC mode plays a huge role in the TLS ecosystem, as it is generally widespread due to historical reasons. Since the CBC mode in TLS can potentially be affected by padding oracle (Section 4.3) vulnerabilities it is important to estimate how prevalent these vulnerabilities are. We, therefore, conducted an extensive study on padding oracle vulnerabilities with the goal of estimating the number and the impact of padding oracle vulnerabilities. To accomplish this, we proceed in three steps. We first define a list of test vectors potentially triggering observable differences which result in padding oracles. We then reduce this test vector list and perform a large-scale scan. Finally, we analyze the identified vulnerabilities and forward the discovered information to the responsible vendors.

9.3.1. Methodology

In our study, we use the `PaddingOracleProbe` described in Subsection 8.3.2. One of our major goals is to notify vulnerable vendors. For every host, we try to gather its *response map*. The response map describes the host behavior when responding to our test vectors. The response map consists of *cipher suite fingerprints*. A cipher suite fingerprint describes the server response behavior for a specific cipher suite and TLS version. A secondary goal for this scan, is

to optimize the scanning process itself, by only using scanning wiht test vectors that are really necessary to find real padding oracle vulnerabilities. We therefore used an empirical test vector reduction approach with a preliminary scan to reduce the number of test vectors without lowering the detection rate. For this purpose, we sampled 50,000 random IPv4 hosts which respond on port 443. We then performed a full scan on these hosts with the aforementioned 25 malformed records and all supported cipher suites and TLS version combinations. We can then analyze our test vector combinations and create the smallest set of test vectors detecting all padding oracle vulnerabilities. These empirical steps ensure that 1) with high probability, we do not miss vulnerabilities, and 2) we can use the reduced set for large-scale analyses. This test vector reduction process resulted in a reduction from 25 to 4 total vectors. This approach is detailed in Section 9.3.

To ultimately find new vulnerabilities we perform our scan one of the *Internet top lists* which typically contain a good mixture of up-to-date server implementations. Among Internet top lists, the Alexa Top 1 Million dataset contains the most significant number of hosts responding to TLS connections (about 75%) and is recommended for TLS scans [162].

After performing the TLS scan with a reduced vector set, we create a list of vulnerable hosts. We re-scan these hosts with our full test vector list.

For the scans, we used a machine with 2 Xeon E5-2683v5 CPUs (with a total of 64 cores) and 48 GB of RAM. The scan used an average of 5Mbit/s of upstream data and 15Mbit/s of downstream data.

9.3.2. Pre-Scanning with All Malformed Records

We performed the preliminary scan with 50,000 random TLS hosts in October 2018 and required three days. The results of this preliminary scan confirmed that the choice of key exchange algorithm and protocol version indeed affects whether a given host exhibits CBC padding oracle vulnerabilities. We then reduced the set of malformed records. To do this, we first identified all vulnerable hosts, i.e. hosts that would be identified when scanning with the full set of malformed records. We then examined subsets of malformed records of increasing sizes, and for each subset, examined the number of hosts that would be identified when scanning only with this subset of malformed records. This process was stopped when a subset of four malformed records identified all vulnerable hosts. That is, all hosts that would be identified when scanning with the full set of malformed records, would also be identified when scanning with the reduced set of malformed records. This reduced set includes the following malformed records (all of these records are 80 bytes in length):

1. A record with missing MAC and correct padding (of value `0x4F`).

2. A record with missing MAC and incorrect padding (of value `0xFF`).

3. An empty record with no application data, with invalid padding and valid MAC. The highest bit in the first padding byte is flipped.

4. An empty record with no application data, with valid padding and invalid MAC. The lowest bit in the first MAC byte is flipped.

Please note that we still test every TLS host with all of its supported cipher suites and TLS protocol versions.

Is the malformed record set reduction lossy? The reduced malformed record set detects all vulnerabilities detected by the larger, original malformed record set, *on the sample data of the preliminary scan.* It is natural to ask whether there are hosts that are vulnerable to a malformed record from the original set, but not to a malformed record from the reduced set. There are obviously no such hosts in the sample data, but there could be such hosts outside of the sample. If there is a large number of such hosts on the Internet, then the malformed record reduction process would be lossy, i.e. by using fewer malformed records, we detect fewer vulnerabilities in the full scan. As we now explain, this source of scanning inaccuracy is likely small enough to not materially affect our results. Put another way, the reduced set of malformed records likely detects most vulnerabilities triggered by the full set of malformed records, not just on the sample data.

Indeed, let p denote the percentage of hosts, out of all TLS-speaking hosts, that are vulnerable to one malformed record from the full set of malformed records, but not to any malformed records from the reduced set. In other words, p describes the percentage of hosts that the reduction misses; we will now show it is rather small. In the random sample of $N = 50000$ hosts used for the preliminary scan, we did not encounter any such hosts. To compute the 99% confidence interval, we require $(1 - p)^N = 0.01$. Solving for p, we obtain $p = 0.0092\%$. We, therefore, determine with 99% confidence that there are at most 0.0092% additional vulnerable hosts that our scans miss due to the malformed record reduction.

We provide an intuitive explanation of the above, for the reader's convenience. As per the above calculation, we estimate the percentage of vulnerable hosts on the Internet that would be missed because we scan with the reduced set of malformed records is 0.0092%. Censys [157] estimates there are about 42.4 million hosts which serve TLS on port 443 as of February 2019. Therefore, our estimate is that the reduction misses at most $42400000 \cdot 0.0092\% = 3900$ hosts. Intuitively, the term "99% confidence interval" means there is roughly a 1% chance that this estimate is wrong, i.e. that there are more than 3900 such hosts on the Internet.

9.3.3. Alexa Top Million Scan

We used the reduced set of malformed records to scan the Alexa Top Million websites. Among the top lists, Alexa Top 1 Million provides the highest percentage of hosts supporting TLS [162] and is thus suitable for large-scale TLS scans. The list likely includes most high-profile TLS implementations.

The scan required approximately 72 hours. Of the initial one million hosts, 785,295 responded on port 443. We were able to perform TLS handshakes with CBC cipher suites with 627,493 hosts. We excluded all other hosts from the

evaluation. We discovered a total of 18,257 Alexa Top Million hosts (1.83%) which are vulnerable to padding oracle attacks.

The data supports our conjecture that implementations may be vulnerable on a cipher suite with one protocol version, but not vulnerable on the same cipher suite with a different protocol version. A total of 649 servers were only vulnerable in either TLS 1.0 or TLS 1.1/1.2 although the vulnerable cipher suite was supported in the other version. Similarly, in some cases, the negotiated key exchange algorithm affects whether implementations exhibit a CBC vulnerability. 601 hosts were vulnerable on one cipher suite, but not on another cipher suite with a different key exchange algorithm but the same symmetric cipher and HMAC function. A total of 3,247 hosts were vulnerable on all CBC cipher suites they supported.

After identifying vulnerable hosts, we rescanned them with the full set of test vectors to get their full response maps. As noted above, to label a host as vulnerable we require the response maps to be consistent across three different scans.

9.3.4. Results of Our Clustering Approach

Analyzing each vulnerable host manually is infeasible. We, therefore, clustered the vulnerable hosts, such that hosts exhibiting the same cipher suite fingerprints are clustered together. This minimizes the manual work required to identify the vendor (or vendors) responsible for each vulnerable behavior.

We identified 93 different cipher suite fingerprints. Table 9.3 summarizes the 40 most common cipher suite fingerprints. Using the first row as an example, 7297 hosts responded with `BAD_RECORD_MAC` and `CLOSE_NOTIFY` TLS alerts and timed out the connection for malformed records 11 and 12 (⏲). For all other malformed records these hosts closed the TCP connection (🚫) after sending the same TLS alerts.

We also identified four groups exhibiting behavior similar to the CVE-2016-2107 vulnerability in OpenSSL [15] (cipher suite fingerprints #41, #75, #14, and #54 in Table 9.3). They respond to malformed records 6 and 7 (see Table 8.3) with a `RECORD_OVERFLOW` TLS alert. To all other malformed records they respond with `BAD_RECORD_MAC`. These are likely unpatched OpenSSL implementations, or security appliances running older OpenSSL versions.

For vulnerable cipher suites on the same host, cipher suite fingerprints are largely consistent. Of hosts exhibiting at least one vulnerable cipher suite, 99.6% have an identical cipher suite fingerprint on all vulnerable cipher suites. We removed the remaining 0.4% of hosts to make clustering easier. However, hosts sharing the same cipher suite fingerprint on vulnerable cipher suites do not necessarily share the same implementation. As an example, consider two hosts, A and B, with two cipher suites supported by both hosts, 1 and 2. A is vulnerable on cipher suite 1 with cipher suite fingerprint X, but is not vulnerable on cipher suite 2. B is not vulnerable on cipher suite 1, but is vulnerable on cipher suite 2 with the same cipher suite fingerprint X. This difference indicates the hosts do not share the same implementation, as we would expect the shared implementation to have a consistent set of vulnerable cipher suites. (We con-

Nr.	Cipher suite fingerprint									Strength		Count
	1,2,3, 20,21	4,5	6	7	8,9	10,16,19, 22–25	11,12	13,14,15	17,18	R1	R2	
15	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W ⏲	F_{20} W 🚫	F_{20} W 🚫	👁	S	7297
41	F_{20} 🚫	F_{20} 🚫	F_{22} 🚫	F_{22} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	👁‍🗨	W	4387
84	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	F_{20} ϟ	👁	P	2313
75	F_{20} W 🚫	F_{20} W 🚫	F_{22} W 🚫	F_{22} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	👁‍🗨	W	940
21	F_{80} 🚫	F_{80} 🚫	F_{20} 🚫	F_{20} 🚫	F_{80} 🚫	F_{80} 🚫	F_{80} 🚫	F_{80} 🚫	F_{80} 🚫	👁‍🗨	W	687
23	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} ⏲	F_{20} W 🚫	F_{20} W 🚫	👁	W	458
68	🚫	🚫	🚫	🚫	⏲	🚫	🚫	🚫	⏲	👁	P	248
0	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	A 🚫	👁	P	194
79	🚫	🚫	🚫	🚫	🚫	🚫	F_{20} W ⏲	🚫	🚫	👁	W	151
10	F_{40} 🚫	F_{40} 🚫	F_{20} 🚫	F_{20} 🚫	F_{40} 🚫	F_{40} 🚫	F_{40} 🚫	F_{40} 🚫	F_{40} 🚫	👁‍🗨	W	98
85	🚫	🚫	🚫	🚫	⏲	🚫	🚫	🚫	A 🚫	👁	P	83
2	ϟ	ϟ	⏲	F_{20} ⏲	F_{20} ⏲	F_{20} ⏲	⏲	F_{20} ⏲	F_{20} ⏲	👁	S	76
61	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	ϟ	ϟ	ϟ	F_{20} 🚫	ϟ	ϟ	👁‍🗨	S	54
6	ϟ	ϟ	ϟ	ϟ	F_{40} 🚫	ϟ	ϟ	ϟ	F_{40} 🚫	👁	P	52
62	⏲	⏲	⏲	F_{20} ⏲	F_{20} ⏲	F_{20} ⏲	⏲	F_{20} ⏲	F_{20} ⏲	👁	S	47
33	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	⏲	👁	P	43
31	🚫	🚫	🚫	🚫	⏲	⏲	🚫	⏲	⏲	👁	P	36
76	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	⏲	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	⏲	👁	P	34
77	F_{20} W 🚫	F_{50} W 🚫	F_{50} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	👁‍🗨	S	28
14	F_{20} ϟ	F_{20} ϟ	F_{22} ϟ	F_{22} ϟ	F_{20} ϟ	F_{20} ϟ	F_{20} ϟ	F_{20} ϟ	F_{20} ϟ	👁‍🗨	W	24
24	F_{20} W ϟ	F_{20} W ϟ	F_{20} W ϟ	F_{20} W ϟ	F_{20} W ϟ	F_{20} W ϟ	F_{20} W ⏲	F_{20} W ϟ	F_{20} W ϟ	👁	W	21
38	F_{80} 🚫	F_{80} 🚫	F_{80} 🚫	ϟ	ϟ	ϟ	F_{80} 🚫	ϟ	ϟ	👁	S	19
4	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	F_{20} ⏲	👁	P	15
54	F_{20} ⏲	F_{20} ⏲	F_{22} ⏲	F_{22} ⏲	F_{20} ⏲	F_{20} ⏲	F_{20} ⏲	F_{20} ⏲	F_{20} ⏲	👁‍🗨	W	12
74	F_{20} W 🚫	F_{20} W ⏲	F_{20} W ⏲	F_{20} W 🚫	F_{20} W 🚫	F_{20} W 🚫	F_{20} W ⏲	F_{20} W 🚫	F_{20} W 🚫	👁	W	9
7	ϟ	ϟ	ϟ	ϟ	🚫	ϟ	ϟ	ϟ	🚫	👁	P	8
37	F_{20} 🚫	F_{50} 🚫	F_{50} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	👁‍🗨	W	7
51	🚫	🚫	🚫	🚫	⏲	🚫	🚫	🚫	🚫	👁	W	7
59	A 🚫	ϟ	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	👁	S	7
66	ϟ	ϟ	⏲	ϟ	⏲	ϟ	⏲	⏲	ϟ	👁	W	7
70	A 🚫	A 🚫	⏲	F_{20} ⏲	F_{20} ⏲	F_{20} ⏲	⏲	F_{20} ⏲	F_{20} ⏲	👁	S	7
11	F_{20} 🚫	⏲	⏲	⏲	⏲	⏲	F_{20} 🚫	⏲	⏲	👁	P	5
42	F_{20} 🚫	F_{21} 🚫	F_{21} 🚫	F_{21} 🚫	F_{21} 🚫	F_{21} 🚫	F_{20} 🚫	F_{21} 🚫	F_{21} 🚫	👁‍🗨	S	5
89	ϟ	ϟ	⏲	🚫	🚫	🚫	⏲	🚫	🚫	👁	S	5
3	⏲	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	⏲	F_{20} 🚫	F_{20} 🚫	👁	S	4
26	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{10} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	👁‍🗨	W	4
28	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	⏲	F_{20} 🚫	F_{20} 🚫	F_{20} 🚫	AW 🚫	👁	P	4
35	ϟ	ϟ	ϟ	ϟ	F_{20} W 🚫	ϟ	ϟ	ϟ	F_{20} W 🚫	👁	P	4
73	🚫	F_{80} 🚫	F_{80} 🚫	🚫	🚫	🚫	🚫	🚫	🚫	👁	W	4
9	F_{20} ⏲	F_{20} ⏲	F_{20} ⏲	F_{20} ⏲	F_{20} ⏲	F_{20} ⏲	F_{20} 🚫	F_{20} ⏲	F_{20} ⏲	👁	W	3

Table 9.3.: Analysis of the 40 most common cipher suite fingerprints, each consisting of responses to 25 malformed records. For ease of reading, we group together malformed records for which responses are identical within each cipher suite fingerprints. We use the following notation: `Application` message (A), Fatal Alert with error code k (F_k), Warning Alert (W), connection closed (🚫), TCP reset (ϟ), timeout (⏲). We use the following TLS Alert codes: `UNEXPECTED_MESSAGE` (10), `BAD_RECORD_MAC` (20), `DECRYPTION_FAILED_RESERVED` (21), `RECORD_OVERFLOW` (22), `DECOMPRESSION_FAILURE` (30), `HANDSHAKE_FAILURE` (40), `ILLEGAL_PARAMETER` (47), `DECODE_ERROR` (50), `DECRYPT_ERROR` (51), `INTERNAL_ERROR` (80). Alerts with code `CLOSE_NOTIFY` always used the `warning` level. 🔒 denotes an encrypted response. The oracle *strength* definition is provided in Subsection 9.3.5; observable differences are depicted with 👁, unobservable differences with 👁‍🗨. We use W and S for weak and strong padding oracles, respectively (a strong and observable oracle is exploitable). P represents behavior similar to POODLE (which is also exploitable if it is observable).

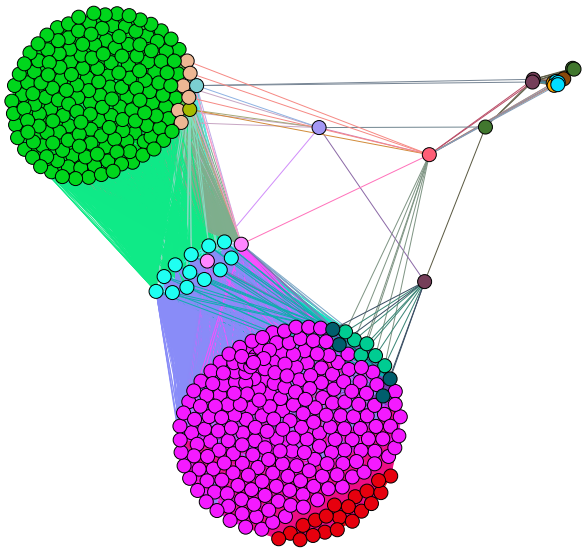


Figure 9.1.: Visualisation of group #23 from Table 9.3.

cede that it is possible the hosts exhibit different behavior because of different configuration flags despite sharing the same implementation, but we consider this unlikely).

We denote the above situation (in its general form) as 'contradictory response maps'; two hosts exhibiting the same cipher suite fingerprint on vulnerable cipher suites, but where there exists a cipher suite supported by both hosts such that one host is vulnerable on that cipher suite and the other host is not. We refer to the complement situation as 'compatible response maps'.

We then use a graph algorithm to further split host groups. For each group of hosts with an identical cipher suite fingerprint, we construct a graph where each node represents a host. We draw an edge between two hosts if and only if their response maps are compatible. We then embed the graph in a two-dimensional plane using the ForceAtlas2 algorithm, as implemented in the Gephi software.[3] The ForceAtlas2 algorithm clusters together nodes connected by an edge, so nodes with compatible response maps are clustered together. Identically configured servers that also behave identically will be connected to the same nodes and therefore have the same degree. Since these servers are connected to the same nodes, ForceAtlas2 will draw them close to one another. By coloring the nodes by their degree it becomes easy to manually spot similarly configured and identically behaving implementations in the graph. These subgroups can then be examined for candidates for manual analysis and responsible disclosure.[4]

[3] `https://github.com/gephi/gephi`

[4] We note that further grouping by the server agent string could provide more insights into the different groups. However, it is also very likely that it would falsify our results. In many cases, TLS is terminated in reverse proxies or firewalls, and the server agent string is generated on a different machine handling HTTP traffic.

Example Vulnerability Group. An example of this visualization is provided in Figure 9.1. The figure clearly shows two distinct sub-groups that do not share edges (meaning their response maps are contradictory and likely do not share the same implementation). Hosts shown in green are vulnerable on `TLS_RSA_WITH_AES_128_CBC_SHA` and `TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA`, while servers shown in pink are only vulnerable to `TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA` and not on `TLS_RSA_WITH_AES_128_CBC_SHA`. Interestingly the hosts in the middle of the graph (mostly in teal) do not support `TLS_RSA_WITH_AES_128_CBC_SHA` (they are vulnerable on `TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA`). They may share their implementation with either the green or pink group and therefore share edges with the members of both groups. Hosts in red are very similar to the pink group but do not share edges with the teal group. This means that either a third group exists, or the teal group actually belongs to the green group and the red group belongs to the pink group. Individual nodes are likely rare configurations of one of the implementations of the bigger groups. We performed a DNS lookup and determined both groups are operated by a Czech hosting company.

This approach allowed us to also contact other prominent websites in each group and ask what TLS implementation they use.

Breakdown of response maps. Figure 9.2 visualizes the prevalence of the various cipher suite fingerprints. A few very common vulnerabilities account for the majority of vulnerable hosts. The newly-discovered vulnerabilities in Amazon/OpenSSL and Citrix account for slightly more than half of all vulnerable hosts. These are listed as #15 and #84 and described in more detail in Subsection 9.3.8. In addition, response maps #41 and #75, which likely stem from implementations based on unpatched OpenSSL versions, account for roughly a third of vulnerable hosts. Response map #23 is found in the above-mentioned Czech hosting company.

9.3.5. Eploitability Evaluation

Not all of the oracles we identified enable effective decryption attacks. The padding oracles we discovered are based on direct message side channels, i.e. on TLS implementations where two error states trigger different error responses from the TLS server. They may be exploitable in the BEAST attacker model, which relies on two assumptions:

- (a) the victim client visits a website under the attacker's control, which triggers HTTPS requests to the victim server, and
- (b) the attacker is a MitM and can observe the session and modify transmitted ciphertexts.

In addition to those standard assumptions, an oracle is exploitable if it satisfies two additional requirements: *(R1) Observability* and *(R2) Perfect padding distinguishability*.

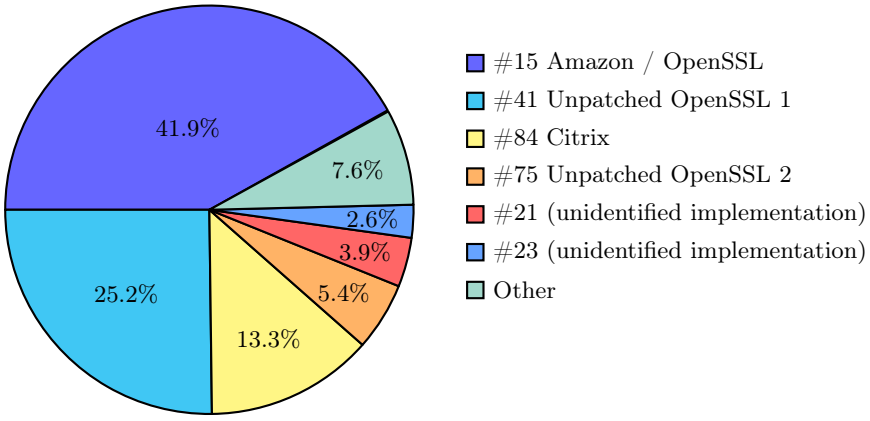


Figure 9.2.: A visualization of the prevalence of cipher suite fingerprints. A few widely-prevalent vulnerabilities account for the majority of vulnerable hosts. Out of the above cipher suite fingerprints, #84 and #15 are exploitable. They are described in more detail in Subsection 9.3.8.

(R1) Observability

Unlike timing side channels, little attention has been paid to direct message side channels in the case of TLS, and common wisdom seems to assume they are *unobservable* to the attacker. Indeed, this is true for implementations which send a single alert in all error cases and the behavior is identical except perhaps for the content of the alert message. Such behavior cannot be exploited by the attacker to create a side channel because the alert message is encrypted. However, we identified many cases where implementations do exhibit an observable difference in behavior. These observable differences can roughly be divided into two classes:

- **TCP layer.** We found implementations which leak information about the padding validity in the TCP layer. For example, in the case of Amazon, most test vectors with invalid padding caused the server to immediately close the TCP connection. However, specific, carefully crafted test vectors caused the server to abort the TLS session while keeping the TCP connection open.

- **Number of TLS records.** We observed TLS servers that responded with a different number of records based on the padding validity. While the attacker cannot decrypt these records, he is able to observe the total ciphertext length. For example, the servers from group 23 (see Table 9.3) responded with *one* TLS alert in the case of valid padding, while for invalid padding they responded with *two* TLS alerts.

Consider an attacker $\mathcal{A}$ who can distinguish between the two cases of valid and invalid padding based on the validity of the last padding byte. The attacker

decrypts an HTTPS session cookie as follows:[5]

1. $\mathcal{A}$ lures the victim client to load a web page he controls. This web page contains JavaScript code which sends HTTPS requests to the victim server, with a URL of $\mathcal{A}$'s' choice.

2. $\mathcal{A}$ observes the first TLS handshake and determines if the negotiated cipher suite is vulnerable to padding oracle attacks. If not, he aborts.

3. If a vulnerable cipher suite is used, $\mathcal{A}$ instructs the client to send another HTTPS request, modifying the URL such that the first character of the session cookie is the last byte in cipher block c_i.

4. As a MitM, $\mathcal{A}$ intercepts the ciphertext $(c_1, ..., c_i, ..., c_n)$ and modifies it such that c_i becomes the last ciphertext block, for example by replacing c_n with c_i.

5. Decryption of this last block c_i is a pseudorandom transform, so the padding will likely be invalid, triggering an observable invalid padding error event.

6. In about 1 out of 256 requests, the padding will randomly be valid. When the padding is valid, it is most likely to be one byte in length. The preceding bytes will be parsed by the TLS server as MAC data, and will be invalid with overwhelming probability. In this case, $\mathcal{A}$ observes a valid padding error event, and computes the first character of the HTTPS session cookie as $c_{n-1}[-1] \oplus c_{i-1}[-1]$, where the $[-1]$ operator denotes taking the last byte of a block.

7. $\mathcal{A}$ then prepares another HTTPS URL where the second character of the session cookie is shifted to the last byte of c_i, and starts again with step 3.

(R2) Perfect Padding Distinguishability

In the above example, we considered a simple oracle that allows for distinguishing between valid and invalid padding based on the validity of the last padding byte. However, even when providing different responses, implementations do not necessarily expose such simple oracles. For example, an older OpenSSL version responds with a different alert message only in the specific case of an empty record containing at least two full valid padding blocks [15]. We identified vulnerable implementations that only respond differently to ciphertexts containing several valid padding or MAC bytes. Such vulnerabilities are less likely to be exploitable since using the algorithm above, the attacker would need to perform far more than 256 oracle queries to decrypt each byte. The attacker may be able to overcome this limitation by inserting bytes of his choice directly after

[5] We present here a more general form of the attack, which is also applicable to POODLE-style oracles. This form requires 256 sessions on average in order to decrypt one plaintext byte [63]. For oracles which completely disregard the MAC, there is a faster form which requires 128 sessions on average to decrypt one plaintext byte.

the cookie value. Due to the malleability property of CBC, it is only possible to insert one block of successive chosen data. Therefore, CBC allows for the creation of practical exploits if the number of chosen padding bytes is smaller than the block size.[6]

Therefore, in our impact estimation, we take a conservative approach. To consider a vulnerable implementation as exploitable, we require that it responds with a valid padding event to ciphertexts with at most one block of valid padding. We call such oracles *strong* and refer to other oracles as *weak*. In addition to these two oracles, we consider oracles which do not correctly validate the complete CBC padding and only validate the MAC. We refer to such oracles as *POODLE oracles*. These oracles could also be exploited by applying attacks similar to POODLE.

Column *R2* in Table 9.3 identifies the oracle strength. For example, servers with the second most prevalent cipher suite fingerprint (#41) respond to malformed records #6 and #7 from Table 8.3 with a `RECORD_OVERFLOW`. In all other cases, the servers send the `BAD_RECORD_MAC` alerts. We consider this group to be weak since the attacker needs to send more than one block of valid padding to trigger the `RECORD_OVERFLOW` alert with a malformed record #6 or #7.

We consider servers with cipher suite fingerprint #2 to be strong oracles. The servers from this group respond with a TCP connection reset (🗲) if they receive a malformed record with a valid padding (see malformed records #20 and #21). There are also several groups with behavior similar to POODLE. These groups ignore modifications in the MAC bytes and respond differently to malformed records #8, #9, #17, and #18.

9.3.6. Exploitability

We consider observable POODLE and observable strong oracles as exploitable. We consider all other oracles as non-exploitable. However, note that weak oracles may be exploitable using more advanced techniques. Our estimate of the number of exploitable hosts is, therefore, a conservative lower estimate.

Estimation of Exploitable Hosts. Our scan identified 18,257 hosts vulnerable to padding oracle attacks. Of those, 11,225 (61.4%) exhibit observable vulnerabilities that allow an attacker to distinguish between two malformed records. See also column *R1* in Table 9.3. At least 10,688 hosts provided strong or POODLE-styled oracles, which is 58% of vulnerable hosts. See also column *R2* in Table 9.3. In total, 10,501 hosts are practically exploitable, i.e. they meet both requirements.

Are CBC cipher suites negotiated? Most modern browsers support AEAD cipher suites. If a vulnerable server prefers AEAD cipher suites, they would likely

[6] Decrypting parts of the cookies with weak oracles or exploiting weak oracles could also be possible with extended techniques. We do not analyze the exploitability of these more complex oracles. Such an analysis would likely need to be done manually for each oracle and would need to consider specific browser behaviors.

be negotiated, and this precludes CBC attacks. 31,651 hosts or 4.03% only support RC4 or CBC cipher suites. Most modern browsers have disabled support for RC4 cipher suites due to [163], so modern browsers would likely negotiate CBC cipher suites with these hosts. Of those hosts, 1,400 were vulnerable to padding oracle attacks.

9.3.7. Ecosystem Insights

In this section, we review our assumptions and present notable vulnerabilities we found in different implementations. We performed our scans under the assumption that scanning with different cipher suites and protocol version is necessary in order to detect vulnerable hosts. As explained below, our findings confirm this assumption.

Is scanning with different protocol versions necessary? Böck et al. [89] found that some servers exhibit RSA padding oracle vulnerabilities only on some of the protocol versions they support. As noted in Section 4.5, we suspected the same holds for CBC padding vulnerabilities. Our findings confirm this assumption: We identified at least 744 hosts that support the same cipher suite in both TLS 1.0 and 1.2, but are vulnerable when using that cipher suite only in one of those versions. In some cases the vulnerable protocol version is the newer version, and in other cases, the older one. As an example of the former case, vine.co was vulnerable using TLS 1.2 with the `TLS_RSA_WITH_3DES_EDE_CBC_SHA` cipher suite, but was not vulnerable when using the same cipher suite in TLS 1.0.

Surprisingly, when only one protocol version is vulnerable with the same cipher suite, there are more cases where the newer version is vulnerable. Out of those 744 hosts, 120 hosts are vulnerable in TLS 1.0 but not in TLS 1.2, and 624 are vulnerable in TLS 1.2 but not in TLS 1.0.

Is scanning with different cipher suites necessary? Böck et al. [89] also found that scanning with different cipher suites is necessary to detect as many vulnerabilities as possible. In the above work, this finding held even when scanning with cipher suites using different symmetric ciphers, while the vulnerability was in the (theoretically unrelated) RSA implementation. We find similar behavior in our results. We identified at least 601 hosts with two cipher suites, one vulnerable and one secure, where the only difference between the two cipher suites is the *key exchange algorithm*. This finding is unintuitive, as one would expect an implementation to be uniformly vulnerable or secure on all cipher suites with the same symmetric cipher. To give one example, one website is secure when using `TLS_RSA_WITH_AES_256_CBC_SHA256` with TLS 1.2, but is vulnerable when using `TLS_DHE_RSA_WITH_AES_256_CBC_SHA256`, also with TLS 1.2.

Rationale Behind the Server Behaviors. Both behaviors may seem unintuitive but are actually expected. Many implementations take completely different code paths depending on the negotiated cipher suite or protocol version. These code paths may, for example, rely on hardware acceleration or use an optimized assembly implementation when possible. It is therefore likely (and, as we see,

common) to find implementations that exhibit vulnerabilities only in some of the supported cipher suites and protocol versions, even when the same symmetric cipher is used.

9.3.8. Notable Vulnerabilities

In our scans we identified multiple devices from Cisco, two different IBM servers, and multiple devices from Sonicwall and Oracle. In the following, we describe specific vulnerabilities we identified and responsibly disclosed in Citrix, OpenSSL, and IBM servers.

Our disclosure is still an ongoing process. Our recent findings and the current state of countermeasures implemented by affected vendors are summarized on `https://github.com/RUB-NDS/TLS-Padding-Oracles`.

Amazon/OpenSSL. With the help of the Amazon security team, we identified a vulnerability (cipher suite fingerprint #15) which was mostly found on Amazon servers and Amazon Web Services (AWS). Hosts affected by this vulnerability immediately respond to most records with `BAD_RECORD_MAC` and `CLOSE_NOTIFY` alerts, and then close the connection. However, if the hosts encounter a zero-length record with valid padding and a MAC present, they do not immediately close the TCP connection, regardless of the validity of the MAC. Instead, they keep the connection alive for more than 4 seconds after sending the `CLOSE_NOTIFY` alert. This difference in behavior is easily observable over the network. Note that the MAC value does not need to be correct for triggering this timeout, it is sufficient to create valid padding which causes the decrypted data to be of zero length. Therefore, we classify this as a strong oracle which is also exploitable.

Further investigations revealed that the Amazon servers were running an implementation which uses the OpenSSL 1.0.2 API. In some cases, the function calls to the API return different error codes depending on whether a MAC or padding error occurred. The Amazon application then takes different code paths based on these error codes, and the different paths result in an observable difference in the TCP layer. The vulnerable behavior only occurs when AES-NI is not used.

We had in fact previously tested the vulnerable OpenSSL code manually, in lab settings, but had not identified this vulnerability. This is because the vulnerability only manifests under a combination of specific conditions: subtle interactions between OpenSSL and external code, and only when AES-NI is not used, which is rare nowadays. We view this as an illustrative example of the usefulness of large-scale scans in detecting vulnerabilities that lab tests may sometimes miss.

We suspect this OpenSSL behavior underlies a number of similar vulnerabilities we identified, not only cipher suite fingerprint #15. Therefore, we hope that once OpenSSL releases a patch, other vulnerabilities will be fixed as a result. The issue was assigned CVE-2019-1559.

The IBM Vulnerabilities. We found multiple vulnerabilities in servers hosted by IBM. One of the vulnerabilities is described by cipher suite fingerprint #77 in Table 9.3. Affected servers respond with a `BAD_RECORD_MAC` alert if either the MAC or the padding is incorrect. If the padding is correct and the MAC is incomplete or not present, the server responds with a `DECODE_ERROR` alert. The latter behavior occurs even if the records are too short to contain a MAC, as long as the record contains at least two blocks of ciphertext, independently of the used MAC algorithm. An attacker can send only two blocks with an IV, which guarantees there is not enough room for a MAC. This provides the attacker with a classic CBC padding oracle. We therefore consider this a strong oracle. Since the alerts are encrypted, we classify this vulnerability as unobservable, and the oracle is therefore not exploitable.

The IBM security team decided to disable CBC cipher suites on the affected servers and to only support AES-GCM.

Citrix. The described vulnerability is identified by cipher suite fingerprint #84 in Table 9.3. The vulnerable implementation first checks the last padding byte and then verifies the MAC. If the MAC is invalid, the server closes the connection. This is done with either a connection timeout or an `RST`, depending on the validity of the remaining padding bytes. However, if the MAC is valid, the server checks if all other remaining padding bytes are correct. If they are not, the server responds with a `BAD_RECORD_MAC` and an `RST` (if they are valid, the record is well-formed and is accepted). This behavior can be exploited with an attack similar to POODLE. Since the oracle is also observable, we consider this group as exploitable. We first detected this vulnerability in Amazon Web Services. In cooperation with the Amazon security team, we determined that Citrix Application Delivery Controller (ADC) and NetScaler Gateway are responsible for this behavior. The vulnerability was assigned CVE-2019-6485.

9.4. Conclusion

The three presented studies have clearly shown the power of tools like TLS-Attacker, TLS-Scanner and TLS-Crawler. It was comparably easy to conduct very complex scanning strategies as demonstrate the the CBC padding oracle study in Section 9.3. This complex study could be conducted in roughly a month, including the development time for the probes and the evaluations scripts. In regards to the ecosystem there are several lessons to be learned.

Raccoon. The evaluation shows that a non-negligible amount of servers are reusing ephemeral keys, and therefore are generally vulnerable to the Raccoon attack. Over 66% of those servers use keys that can be exploited by us with a bit leak of $k = 8$, while the remaining servers require more bits to be leaked in order to practically solve the HNP with our approach. The data also shows that direct oracles ($\mathcal{O}^{\mathrm{D}}$) are found in real TLS implementations, which lifts the Raccoon attack attack from a hard to exploit timing attack, to an easy exploitable vulnerability.

Alpaca. The evaluation of the Alpaca attack has shown the general attack scenario is widespread on the Internet. While we only scanned a limited number of ports, we identified over 1,4 million hosts that used a compatible certificate to an HTTPS server. This clearly shows that the attack is not a theoretical issue. Furthermore, the evaluation revealed that SNI and ALPN were no severe obstacles at the time because they were not perceived as security tools and, therefore, not strictly evaluated.

CBC Padding Oracle. This study demonstrates that padding oracle vulnerabilities still exist on the modern Internet and will likely continue to threaten users' security. These vulnerabilities are often hard to detect: they may rely on subtle side channels or require specifically-crafted inputs in order to trigger.

In the past, major new TLS attacks had positive effects on the ecosystem. For example, the work by Adrian et al. [115] resulted in an 'enforcement' effort, where major browsers changed their behavior and refused to connect to servers with weak DH parameters. It is an interesting open question how the security community can better help server operators detect and remediate more subtle kind of vulnerabilities (CBC oracles in particular, and other classes of vulnerabilities in general).

One solution in the context of CBC oracles would be to disallow CBC cipher suites altogether. Recently, major browser vendors have declared their intention to remove support for the old 1.0 and 1.1 TLS versions. This forces many server operators to upgrade their implementations or change configuration. Indeed, a case could be made that browser vendors can also remove support for CBC cipher suites, forcing again server operators to upgrade. These changes are not without their costs; they usually require notice of months in advance, may require coordination between browser vendors, and obviously, create additional work for server operators.

General Lessons. Our results confirm that large-scale scans make it feasible to uncover a large variety of security vulnerabilities, previously not detected by lab testing. We believe that this approach is of general interest not only in the context of TLS. One open question is how to identify vulnerable implementation versions and their vendors. In the SSH and IPsec protocols, these data are typically transmitted as message fields in the protocol. Transmitting such data in TLS would make disclosure easier, but on the other hand would lead to privacy issues and easier fingerprinting.

10

Fuzzing TLS Implementations

Since TLS implementations are often implemented in memory-unsafe languages like C or C++, they are also prone to typical memory and integer overflow attacks. One of the most well-known attacks belonging to this category is Heartbleed, which allowed remote attackers to steal server private keys [152].

The steadily increasing importance, combined with continually appearing attacks rooted in the complexity of TLS implementations, has motivated researchers to search for new methods to evaluate the security of TLS libraries. *Fuzzing* is a technique to automatically test software for its robustness [164]. The main idea of fuzzing is to produce randomly or carefully modified, possibly invalid inputs that trigger specific corner cases in the tested application, such as application crashes, buffer boundary violations, or integer overflows. If the tested application fails to handle such a malformed input, the fuzzing framework detects this behavior and provides the developer with information about the resulting misbehavior.

Over the past years, several different fuzzing strategies emerged, and the quality of fuzzers improved. By design, a classical fuzzing framework can only find bugs in parts of the application that were executed while processing fuzzing inputs. Therefore, one of the main fuzzing goals is to reach as many code parts of the application as possible and thus improve the tested code coverage.

This section presents Evolutionary TLS Fuzzer (ETF), an evolutionary greybox TLS fuzzer. The fuzzer is presented in the paper 'Protocol Aware Greybox Fuzzing of TLS Implementations' which is still under peer review at the time of writing. ETF was developed by me under the direct supervision of Juraj Somorovsky. The evaluation of TLS clients was also conducted by me, while the server evaluation was done be Emre Güller. Philipp Görz implemented the communication synchronization mechanism, while I implemented the mechanism inside ETF. The project was further supervised by Jörg Schwenk and Thorsten Holz.

10.1. Fuzzing

Feedback-driven Fuzzing. While fuzzers can detect a large variety of bugs by feeding mutated inputs into an application until it crashes, these so-called blackbox fuzzers are inherently limited by the quality of the provided seed files and thus cannot reach deeper code regions on their own. The feedback-driven approach to fuzzing is a significant improvement as it introduces an evolutionary algorithm approach. The idea behind feedback-driven fuzzing is to inspect the evaluated application and collect information about effective input modifications. There are different metrics that can be used to guide the fuzzer, for example:

- *Function coverage* indicates whether a specific function was executed.
- *Block coverage* measures the number of code blocks hit during the fuzzing process.
- *Branch coverage* measures whether each possible transition from one code block to another was taken. Note that branch coverage implies block coverage.

Protocol Aware Fuzzing. While providing completely random data to an application can result in a software crash, it is often more likely that the tested application discards the fuzzing input directly, because it expects the input to follow a specific structure. General purpose fuzzers therefore often fall short if the input should contain checksums or cryptographic protection mechanisms like message authentication codes (MACs). If the cryptographic validation fails, the input is discarded and fails to reach relevant parts of the application. Protocol-aware fuzzers bypass this issue since they understand the evaluated protocol and can generate inputs that behave almost properly and reach deeper into the application. Lee et al. [165] provided an example of such a fuzzing approach, which analyzed the security of Software-Defined Networks (SDNs). This is generally referred to as *grammar fuzzers*, i.e. when the SUT requires a structured input.

10.2. Evolutionary TLS Fuzzer

To improve the state-of-the-art in TLS fuzzing, we developed a feedback-driven TLS fuzzing framework called Evolutionary TLS Fuzzer (ETF). Our TLS fuzzer is divided into several interchangeable modules which can be used to create specific fuzzer configurations. The different modules are visualized in Figure 10.1:

- The mutator module defines how the fuzzer manipulates/creates protocol workflows during fuzzing.
- The executor is responsible for the actual execution of the workflow. This module is mainly based on the TLS-Attacker library.

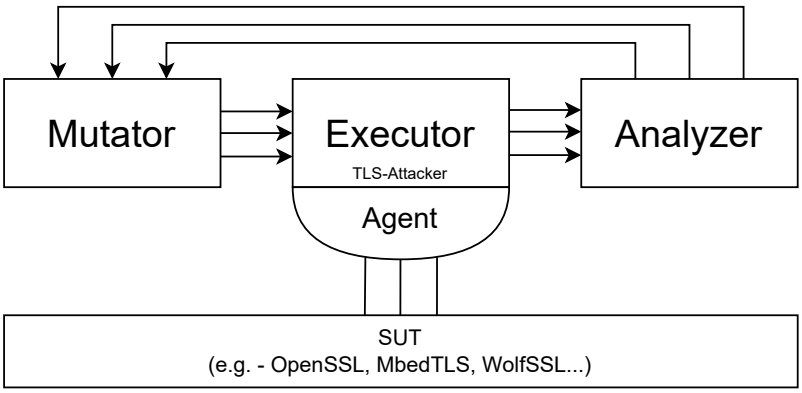


Figure 10.1.: The architecture of our evolutionary protocol aware TLS fuzzer.

- The agent communicates with the fuzzing target. This mainly includes starting and stopping the target and retrieving the instrumentation output.
- The analyzer evaluates the protocol runs and the instrumentation output to provide feedback to the mutator. It is also responsible for the detection of protocol violations.

The whole fuzzing process is multi-threaded to improve performance. First, the mutator generates a test input that contains a TLS protocol flow definition (a `WorkflowTrace`), TLS-Attacker configuration, and a set of parameters used to start the TLS target. It then passes the created test vector to the executor. The executor starts the fuzzing target with the help of the agent and uses TLS-Attacker to execute the provided `WorkflowTrace`. If the agent supports instrumentation, the executor collects the relevant information together with the executed protocol flow. This data is then handed over to the analyzer, which decides whether the executed protocol flow shows interesting or unexpected behavior or whether the instrumentation output shows newly discovered code branches. This design allows one to combine different modules easily and add new ones to create a TLS fuzzer for specific use-cases. In the following sections, we discuss the modules and their most relevant properties and functionalities in more detail.

10.3. Mutator

The mutator module is responsible for the generation of modified protocol flows. While arbitrary mutation strategies can be implemented, we implemented a strategy inspired by parameter optimization for our case study.

Workflow Seeding. In the initial fuzzing phase, the fuzzer seeds itself and generates test vectors with (semi) valid handshakes. Those test vectors are then executed, and the instrumentation output of the target is extracted. If a test

vector reaches a new code branch in the implementation, it is stored in the corpus.

Workflow Mutation. The initial seeding phase is followed by the actual mutation phase, where random test vectors are loaded from the corpus and modified before they get executed again. If the mutated test vector finds a new code branch during execution, the new test vector is again added to the fuzzing corpus. In contrast to general-purpose fuzzers, our inputs are present in an abstract form and are not stored as static byte streams. In this abstract form, the fuzzer can modify the input much more targeted than traditional fuzzers. The mutator can therefore choose to modify the input on a semantic level. We currently implemented the following mutations:

- Add a *ModifiableVariable* to a message/record
- Change a value in the `Config`
- Add a new action to the `WorkflowTrace`
- Add a new flight of actions to the `WorkflowTrace`
- Change the values within an action
- Add message structs to a message
- Change the certificate
- Change the startup parameters of the target
- Splice two inputs together

These mutators allowed us to create a corpus of test vectors that reach different corner cases in a protocol implementation.

Parameter Optimization. While the list of possible mutations seems short, there are many choices to be made by the mutator which can be optimized. For example, suppose the mutator decides to add a modifiable variable to a field in a message. In that case, it is challenging to choose the field, such that the modification will most likely find new code coverage. Concretely, modifying the cipher suite in a `ServerHello` message is more likely to result in additional code coverage than the `verify_data` in a `Finished` message. That is because the cipher suite in a `ServerHello` has potentially hundreds of relevant, semantically different values which influence the code flow. In contrast, the `verify_data` is a cryptographic checksum and thus has only a few semantically different values. If we were to modify both fields with equal opportunity, we would waste many executions on a field that we have quickly (semantically) exhausted. Additionally, even if we chose a field that we want to modify, it is still undecided *how* we want to modify a given field. Sticking to the example of the `verify_data`, we could either flip a bit, delete a byte, add a byte, and so on. Even if we chose a modification type, we still have to choose the exact position within the

field where we want to apply it, since some positions are potentially better for a given modification and in a given field.

A simple solution to these problems is manually choosing reasonable values for all these questions beforehand. While this approach is viable, it suffers from various downsides. First, the optimal parameters for the mutator may change from implementation to implementation. For example, mutating the cipher suite field might be better if the fuzzed implementation supports many cipher suites. However, if the fuzzed implementation supports only one cipher suite, it is better to focus on other fields. The other major problem with this approach is that choosing specific modifications adds biases to the fuzzer and can make the fuzzer unable to find unexpected values.

In this work, we solved this issue by using parameter optimization. At the start of the fuzzer, each potential choice the mutator can make has a probability distribution. Whenever the fuzzer chooses a value from a distribution, the chosen value and the probability range are added to the input as metadata. Once the input is executed, the analyzer can use this metadata to provide feedback to the mutator. If test input reached new code coverage, the mutations were good; therefore, the probability of this kind of choice being made again will increase. The amount of increase depends on the amount of code coverage reached by the test input. A choice that unlocks an entirely new feature for the fuzzer within the implementation is rewarded more than a choice that only finds a new one-line error case. The probability for a given choice can never reach zero, giving the fuzzer always the chance to apply rarely useful mutations. Over time, we expect that for each choice, the fuzzer will find reasonable probability distributions for each parameter without human bias.

Parameter-Map Mutator. Most TLS libraries provide example applications that can be used to test the library. For example, OpenSSL provides the `s_client` and `s_server` applications. These applications often have many command-line parameters that can be used to activate or deactivate specific TLS features. The enabling or interaction between these TLS features within a library can dramatically influence the achieved code coverage.

Our goal was to evaluate most of the provided features without explicitly restarting our fuzzer for each possible configuration. We, therefore, introduced the concept of *parameter maps*. A parameter map is an XML file that describes the possible configuration options of the command line program. These maps are loaded by the fuzzer on startup and are then part of a test vector. The mutator can then decide to activate or deactivate specific parameters. If the mutator occasionally creates a parameter map that causes the fuzzing target to fail to start (e.g., an invalid parameter choice), the test vector is discarded. Using these strategies, we can automatically create different test target configurations and achieve higher code coverage.

Certificate Corpus Mutator. Our fuzzer generally concentrates on evaluating TLS protocol-specific vulnerabilities. It does not specialize in creating arbitrary certificates or certificate fuzzing. Certificate fuzzing can be executed without

protocol-aware fuzzers and has already been extensively studied in previous research [166–171]. Fuzzing certificate formats within a TLS handshake is generally not very efficient as a separate connection has to be established for each test. Since X.509 certificates have already been a target of sophisticated fuzzing research in the past, we allow our fuzzer to load the fuzzing corpora of other fuzzers with malformed certificates that our fuzzer can send during the handshake. Concretely, we use the X.509 corpora within OpenSSL. This created a hybrid approach of previous certificate fuzzers and our ETF, which tests the malformed certificates in the context of real TLS connections, and assists TLS-Attacker in areas where it is not specialized. Please note that in this case, ETF partially loses its protocol awareness since it does not possess the keys for the malformed certificates.

Scope. Overall, our fuzzer supports 47 different messages resulting in a total of 1122 modifiable fields. The `Config` for TLS-Attacker contains 265 fields that the fuzzer considers for modifications. Most of these fields offer the fuzzer multiple points of choice, for example, how long a specific value should be or which type of modification should be applied. Additionally, the fuzzer can decide to add or remove complete actions or sub-structs within messages. This results in over 10000 different points of choice for the mutator making the input space to explore for the fuzzer large.

10.4. Executor

The executor is responsible for the TLS protocol execution based on the test vector generated by the mutator. It first uses the agent to start the system under test with the parameters specified in the test vector. It then takes the `Config` and `WorkflowTrace` from the test vector and utilizes TLS-Attacker to perform a TLS protocol flow with the system under test. Finally, it passes the execution results, like the instrumentation output and the executed `WorkflowTrace`, to the analyzer. There are two different executors implemented, one for the fuzzing of clients and one for the fuzzing of servers.

10.5. Agent

We implemented two agents, one relying on AFL (`AflAgent`) and one relying on unspecific compiled binary (`BlindAgent`). `AflAgent` expects that the binary has been instrumented by AFL. It then starts the fuzzing target and retrieves the instrumentation output via shared memory. Note that the usage of `AflAgent` requires access to the source code of the target. `BlindAgent` does not rely on the source code availability; it simply starts the target binary but does not provide instrumentation output. `BlindAgent` can be a viable option for closed-source implementations if a good corpus was already created beforehand (e.g., by fuzzing other implementations). `BlindAgent` was not used during our evaluation.

10.6. Analyzer

Our analyzer module is responsible for analyzing the instrumentation output, collecting statistics, and deciding if the input resulted in a finding. It accepts the executor output from a TLS protocol run and, therefore, also has access to the complete execution transcript (including cryptographic computations done by the fuzzer).

The analyzer allows the user to define which protocol violations they are searching for. This can include but is not limited to implementation crashes, undefined behavior, or specific TLS protocol violations. Additionally, the analyzer filters duplicate results. This is important since the fuzzer usually runs autonomously for longer periods. The fuzzer likely finds the same issues over and over again. Without filtering, the fuzzer would store the findings repeatedly, making the analysis of discovered issues very slow. Therefore, the analyzer filters duplicate issues based on either their bitmap or a manually curated list of known-issue indicators.

Protocol Violations. Since the analyzer has access to the transcript and the cryptographic computations, it also has the potential to find protocol violations by carefully checking sent and received messages. However, this requires in-depth domain knowledge from the fuzzer about which behavior of the target is acceptable and which behavior is not, effectively requiring the fuzzer to solve the test oracle problem [172]. This is generally considered a hard problem. Detecting protocol violations is especially hard since the fuzzer can perform arbitrary chained modifications. Multiple modifications can, for example, cancel each other out or change the semantics of a message entirely. For example, the fuzzer could choose to remove a byte of a specific field but add the same byte in the next upcoming field, resulting in the same message as if no modifications were applied at all.

Given the complexity of detecting protocol violations, we decided to only detect protocol violations signaled directly by the system under test. To demonstrate such a use case, we implemented the detection of internal errors as returned in TLS alerts which a TLS conforming implementation should never return. According to RFC 5346 [52], internal errors in TLS are defined as follows:

> *An internal error unrelated to the peer or the correctness of the protocol (such as a memory allocation failure) makes it impossible to continue.*

There have been cases where internal errors leaked through alerts did have security implications. For example, the leaked code path has led to Bleichenbacher vulnerabilities [173]. All errors triggered through fuzzing in our fuzzing environment are, by definition, externally triggered and therefore should not trigger *internal errors*, therefore, receiving an internal error is always treated as an interesting finding which should be analyzed further.

10.7. Communication Synchronization

During the fuzzing process, we cannot make assumptions about the behavior of our peer, as our inputs typically violate the specification and can result in additional (unexpected) message exchanges. With a full blackbox approach, it is unpredictable if the peer is already done answering the input or if the peer is still processing it. The fuzzer, therefore, must wait a given timeout for a response after each message it sends. This chosen timeout is critical to the number of handshakes our fuzzer can execute. If we would set this timeout too low, our fuzzer might miss interesting results since we proceeded with our workflow before the peer had the chance to respond. If we set this timeout too high, our fuzzer will waste crucial resources waiting for the peer which already finished processing our input.

We solved this problem by implementing the *communication synchronization* technique similar to *network-emulator* [174]. We load a shared library into the fuzzer target that hooks the interactions with the network layer. Whenever the target tries to read data from the socket without data being available, we notify the fuzzer via named pipes that the target is ready to process more data. This eliminates the need for a timeout as the fuzzer can now directly see when the target has processed the input and can continue the execution immediately.

10.8. Evaluation

The primary contribution of ETF is its ability to reach and trigger bugs in TLS implementations that require extensive knowledge of the protocol without modifications to the code. We evaluate our approach to achieve high code coverage using TLS *servers* and compare the results with three state-of-the-art protocol fuzzers. Using TLS *clients*, we evaluate our approach on the ability to find flaws.

The goal of our evaluation is to answer these research questions:

1. How does ETF perform against other TLS-aware fuzzers in terms of code coverage?

2. How does ETF perform against a handcrafted test suite specifically targeting TLS implementations?

3. Does ETF identify real-world bugs in TLS clients?

Regarding research question (1), Section 10.10 demonstrates that our TLS-aware implementation can cover more branches than comparable fuzzers. For this purpose, we set up three TLS-aware fuzzers:

- AFLnet [16]: a feedback-driven fuzzer based on `AFL` that works for network applications. It provides a TLS protocol state-aware mode, which we use.

- TLS-Attacker 1.2 [15]: The original TLS-Attacker project already contained a simple fuzzer. This allowed the author to find new vulnerabilities

in widely used TLS libraries, including OpenSSL [175], MatrixSSL [176], and Botan [177]. The original fuzzer did not have a feedback mechanism.

- tlsbunny [178]: a Java framework to build fuzzers for TLS 1.3.

Regarding RQ2, Section 10.11 shows that our TLS-aware implementation can reach more code than a hand-crafted selection of scripts to test TLS implementations (a test suite called tlsfuzzer [179]).

To answer RQ3, in Section 10.12 we perform a case study and present bugs ETF was able to find in TLS clients.

10.9. Test Setup

To achieve a clean setup, all fuzzers are built as dockerized images to not only allow easier use (and increase maintainability) but also to have the ability to restore and reset the images before each new evaluation run. We have decided upon a few principles to keep the comparison fair between fuzzers:

1. All TLS libraries are compiled and built only once, the same binary is used by all fuzzers, i.e., the compilation flags do not change between fuzzers.
2. To allow all other fuzzers to reach deeper code paths, the TLS libraries are compiled in fuzzing mode where possible, which sometimes disables some cryptographic checks and uses a fixed seed for randomness.
3. The configuration flags of TLS servers are not dependent on the fuzzer (i.e., they stay the same) - we disable the `Parametermap Mutator` in ETF during the code coverage evaluation to keep the comparison fair and only enable it during our case study to find vulnerabilities.
4. Fuzzers use the same flags (e.g., fuzzer timeout) for all TLS libraries unless specified otherwise.
5. The same RSA 2048-bit key and certificate is configured for all fuzzers.

To measure code coverage, we use a modified version of AFLnet's `afl-showmap` with an increased bitmap size to dump edge coverage information while running the server in the background. This is supported by all fuzzers and test suites and allows us to compare coverage results across all instances. `afl-showmap` is instructed to run the server and dump the cumulative edge coverage information once a minute with a timestamp.

To run the code coverage evaluation, we follow the guidelines by Klees et al [180]. As fuzzing runs are intrinsically random, all experiments are repeated ten times with a runtime of 24h. Additionally, we perform the Mann-Whitney U test for statistical analysis and calculate the p-value to check if the results are statistically significant. We use the median code coverage to reduce the effect of outliers and calculate the overall improvement via the geometric mean.

We have decided against the `ProFuzzBench` benchmark [181] for stateful protocol fuzzing as it only provides one TLS subject (and one DTLS subject).

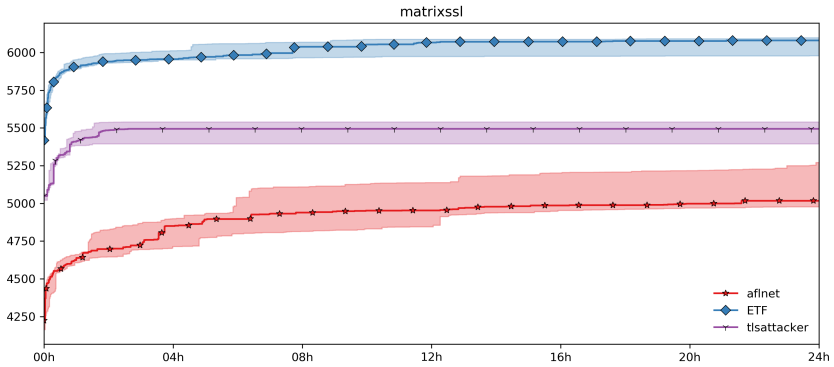


Figure 10.2.: Edge coverage over time for MatrixSSL. The dotted lines show the median run, the colored areas highlight the 95% confidence interval. Y-Axis does not start at zero to allow better visual distinction. Plots for the other TLS libraries are in the appendix.

Instead, we evaluate against these seven TLS libraries: BoringSSL, Botan, GnuTLS, MatrixSSL, mbedTLS, OpenSSL, and wolfSSL. All TLS libraries come with an implementation of server and client, which we use as fuzzing targets. For the specific configuration parameters and version numbers, see Table B.1 in the appendix for the code coverage evaluation, and Table B.2 for the client fuzzing case study.

For the following experiments, we use two machines with 40-core/80-thread Intel Xeon Gold 6230 CPU @ 2.10GHz processors and 192GB RAM. In total, the evaluation required about 10,000 CPU hours.

10.10. Code Coverage - Fuzzing TLS Servers

To answer **RQ1**, i.e., how does our fuzzer compare against other TLS-aware fuzzers, we use the aforementioned code coverage setup and evaluate against three fuzzers: TLS-Attacker, AFLnet, and tlsbunny. As ETF is based on TLS-Attacker, this experiment also demonstrates the effectiveness of our new coverage-guided approach and the speed increase due to communication synchronization.

AFLnet requires seed files for the fuzzing process, similar to `AFL` which it is based on. To generate these seed files for all TLS servers, we follow AFLnet's tutorial on their GitHub page [1]. As all TLS libraries come with their own sample server and client implementations, we use those to generate the seeds as follows: we start `tcpdump` in the background to sniff on the network device, then we start the sample server with the same parameters as used in the evaluation and run the sample client just once to capture the packets. We use `Wireshark` to extract the relevant TCP stream, as described in the tutorial.

Note that AFLnet needs to run without the `-E` flag (thus disabling the state-

[1] `https://github.com/aflnet/aflnet#tutorial---fuzzing-live555-media-streaming-server`

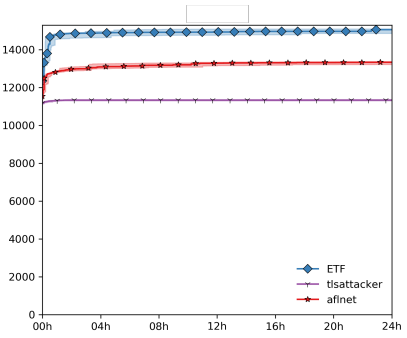
aware mode) for the Botan TLS library, as this otherwise leads to a crash in the fuzzer. This is the only custom modification to the fuzzer flags that is required for the evaluation. tlsbunny is limited in its support of TLS servers (e.g., it only supports TLS 1.3). As such, it is only used on three of the seven TLS libraries (GnuTLS, OpenSSL, wolfSSL). Because tlsbunny is a framework for building fuzzers and not a fuzzer in itself per se, we introduce some modifications to make it work in our evaluation context. First, we use `DeepHandshakeFuzzyClient` and allow RSA as a signature scheme, so it works with the certificates and keys we use for all other fuzzers. Additionally, we disable a self-test requirement, so it runs in our scenario.

Lastly, we had to make minor adjustments to two TLS libraries to make them run with all fuzzers: MatrixSSL was patched with the correct path to our supplied certificate and key files. wolfSSL needs to be started in the root install directory, making it difficult to run directly by the fuzzers, so we patched the source file to allow the binary to be used from any path.

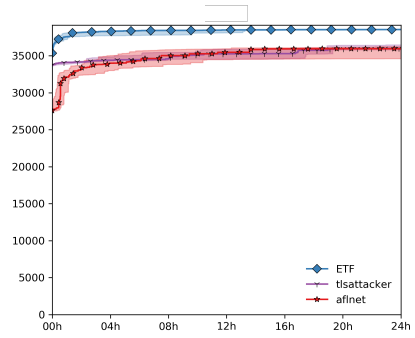
Experiment. We run all dockerized fuzzers on every target ten times for 24h and measure the edge coverage over time. To compare the performances, we take the median of the ten cumulative edge coverages and calculate the relative improvement on a per-binary basis. The overall improvement per fuzzer is calculated with the geometric mean overall relative improvements.

Results. As seen in Table 10.1 and Figures 10.2 and 10.3, ETF outperforms all other fuzzers. Our experiments succeeded with strong, statistically significant results ($p < 0.01$). To keep the comparison fair, the improvement in the table is calculated relative to the second-best fuzzer (AFLnet). As such, ETF comes with at least a $+$**17.7%** improvement to other state-of-the-art TLS-aware fuzzers in terms of average edge coverage. More specifically, on average ETF reaches **24.7%** more edges than TLS-Attacker, and **112.7%** more than tlsbunny. A close inspection of the speedup reveals that the highest coverage the median run of AFLnet can achieve (usually close to the end of the runtime, after 22-24h), the median ETF run can **surpass in less than seven minutes** after the evaluation starts, as seen in Figure 10.3.

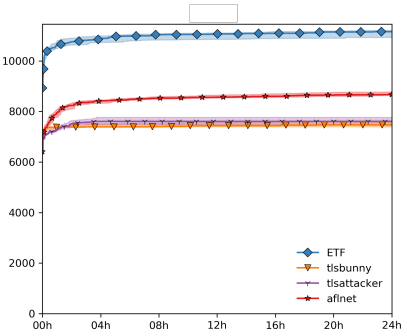
Conclusion. This experiment demonstrates that ETF can reach significantly more code than comparable fuzzers. Note that ETF outperforms AFLnet despite having a lower executions-per-second rate. As such, even more improvements are expected with future performance gains. Additionally, to keep the comparison fair, we have disabled ETF's `Parametermap Mutator` for this evaluation, which would have started the server with different flags to make more code regions reachable. In real-world scenarios, this would allow ETF to gain additional improvements compared to the other fuzzers that do not have this functionality. Our speedup also shows that despite not having a binary-specific seed to start with (like AFLnet), ETF's in-depth knowledge of TLS allows us to cover the same number of edges as the second-best fuzzer in multiple orders of magnitude less time.



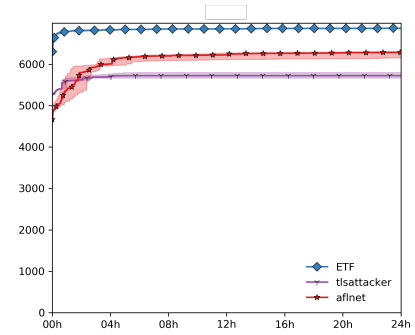
(a) BoringSSL



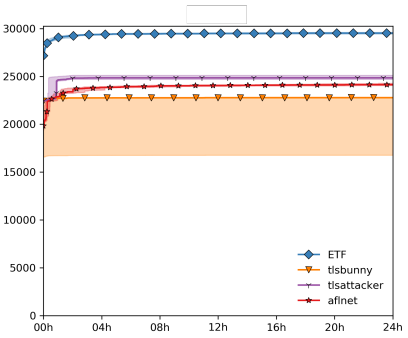
(b) Botan



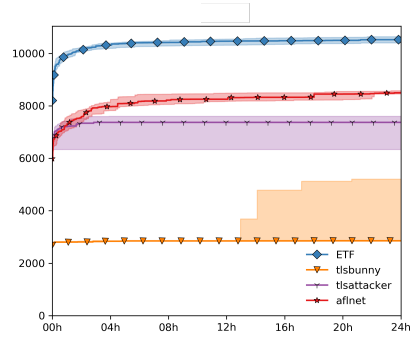
(c) GnuTLS



(d) mbedTLS



(e) OpenSSL



(f) wolfSSL

Figure 10.3.: Edge coverage plots over time for the TLS libraries tested. The dotted lines show the median run, the colored areas highlight the 95% confidence interval.

TLS Library	ETF	AFLnet	TLS-Attacker	tlsbunny
BoringSSL	**15070 (+13.0%)**	13337	11334	-
Botan	**38563 (+7.2%)**	35990	35998	-
GnuTLS	**11165 (+28.7%)**	8675	7608	7474
MatrixSSL	**6084 (+21.0%)**	5027	5495	-
mbedTLS	**6866 (+9.3%)**	6283	5725	-
OpenSSL	**29531 (+22.3%)**	24150	24868	22776
wolfSSL	**10531 (+23.9%)**	8502	7424	2862
Improvement	**+17.7%**	baseline	-5.6%	-35.1%

Table 10.1.: Median edge coverage (10 runs with a run time of 24h). Higher values are better (bold values highlight the best result). The improvement of the fuzzers are calculated as the geometric mean over all per-server relative improvements to AFLnet.

10.11. Code Coverage - Test Suite for TLS Servers

To answer **RQ2**, i.e., how does ETF compare against a test suite, we select tlsfuzzer to evaluate against, which is a list of handcrafted scripts specifically targetting TLS implementations [179]. Its main purpose is to check for correct error handling (i.e., does the implementation return correct error messages). Although some fuzzing functionality exists in a few scripts with randomized inputs, most scripts test a specific, handcrafted input, e.g., one script checks how the server handles zero-byte payloads in the app data records.

Experiment. As tlsfuzzer is not a traditional fuzzer but a collection of scripts, we run all scripts against the TLS server to collect edge coverage information. Because there is no intrinsic order in which the scripts should be executed, and the TLS server might take different edges informed by the state machine, all scripts are executed in a randomized order at every loop to ensure that the test suite captures all possible edges. The experiment is run ten times for 24h and compared against ETF from the previous experiment.

Results. ETF outperforms tlsfuzzer by **+6.0%** in terms of edge coverage averaged over all TLS servers ($p < 0.01$). However, on a per-server basis, ETF can only outperform tlsfuzzer on six of seven binaries.

Conclusion. Although our improvement on the test suite is not large on all binaries, we believe this is not an either-or situation between ETF and tlsfuzzer. In real-world scenarios, a security analyst is advised to combine this test suite with our dynamic fuzzer to cover more code than either one could reach on their own. As such, both solutions are complementary and should be applied as such.

Nevertheless, this experiment demonstrates how a feedback-driven TLS-aware fuzzer can keep up with a hand-crafted collection of scripts targeting specifically known execution paths.

TLS Library	ETF	tlsfuzzer
BoringSSL	**15070 (+6.6%)**	14136
Botan	**38563 (+0.2%)**	38497
GnuTLS	**11165 (+0.1%)**	11159
MatrixSSL	**6084 (+10.3%)**	5517
mbedTLS	**6866 (+7.0%)**	6416
OpenSSL	29531	**29708 (+0.6%)**
wolfSSL	**10531 (+19.9%)**	8786
Improvement	**+6.0%**	baseline

Table 10.2.: Comparison of ETF and tlsfuzzer. Median edge coverage (10 runs with a run time of 24h). Higher values are better (bold values highlight the best result). The improvement is calculated as the geometric mean over all per-server relative improvements.

10.12. Case Study - Fuzzing TLS Clients

We started using ETF to discover bugs in TLS clients in 2017 and were able to identify 25 bugs in 7 libraries in total (see Table 10.3). Since then, we extended the fuzzer and improved its performance. For a quantitative overview see Table 10.3.

Internal Error Alerts. As described in Section 10.6, internal error alerts should not be returned by the TLS peer in a controlled testing setup. If despite the TLS specification, we received internal errors from the TLS client implementation, we treated each such internal error alert as a potential bug.

We found several ways to generate an internal error in GnuTLS resulting from an incorrect mapping from errors to TLS alerts. To give some examples, sending a `CloseNotify` or `ServerHello` with unexpected session ID length will result in an internal error alert sent by GnuTLS.

Similarly, mbedTLS responded with an internal error when receiving an unknown cipher suite in a `ServerHello` message.

OpenSSL reacted with illegal internal errors in six cases, for example, when given a `CertificateRequest` message with missing Signature and Hash algorithms.

Additionally, we found one case of erroneous internal error in BoringSSL.

While such errors are usually not direct security issues, they do violate the standard and are typically used by developers to indicate that something happened that subverted developer expectations. An implementation that sends internal errors when it notices that assumptions in the code are violated greatly assists in discovering possible bugs.

Memory Errors. We searched for buffer overflows and overreads using the address sanitizer model of CLANG [182]. We found multiple buffer overflows, buffer overreads, and null pointer dereferences in various client implementations. We manually investigated each found issue and deduplicated them further manually. We found two cases of null pointer access in MatrixSSL which

TLS Library	Internal Error	Memory Error	Undefined Behavior
BoringSSL	1	0	0
Botan	0	0	0
GnuTLS	2	0	0
MatrixSSL	0	5	3
mbedTLS	1	1	1
OpenSSL	6	0	1
wolfSSL	0	1	3
Total	10	7	8

Table 10.3.: With ETF we were able to find 25 issues in the analyzed libraries in total.

resulted in a segmentation fault and three heap buffer overflows. We found a crash in wolfSSL that can be triggered in the client code by sending a valid *SignatureAndHashAlgorithm* extension in a `ServerHello` message. According to [52] servers MUST NOT send this extension, and the wolfSSL library did not properly deal with receiving that extension in client mode. This bug illustrates the benefit of minimizing human bias as this kind of mutation is easily missed.

An example of a discovered buffer overread can be found in mbedTLS. mbedTLS fails to verify the signature and hash algorithms length field in a `CertificateRequest` message, and unintentionally writes raw memory into the console log. This bug validates the parameter maps mutator used by our fuzzer, as it only appears when the debug mode is enabled. The mutator triggers the issue by automatically changing the client application parameters.

All in all, we discovered seven distinct memory issues which resulted in a crash.

Undefined Behavior. Undefined behavior is elicited by code that is not precisely defined according to the programming language standard. This is an intended feature in C and C++ to allow the compiler to make more optimizations by making certain assumptions about code. Often to allow the code's behavior to depend on the underlying hardware. Violating these assumptions results in undefined behavior and is always an error in actual code, even though the code may work with specific architectures and compilers.

We use the checks available in the CLANG compiler, covering about 20 kinds of undefined behavior [183]. We found several cases of undefined behavior in MatrixSSL, wolfSSL, mbedTLS, and in OpenSSL. They include potentially dangerous use of null pointers, illegal struct member access, or undefined shifting operations. One particularly interesting example of such undefined behavior is the case of OpenSSL. OpenSSL uses the `memcmp` function to compare the subject name with the issuer name in X509 certificates. With a specially crafted certificate, one can trick OpenSSL into using a null pointer as a parameter for `memcmp` which led to unspecified behavior. Interestingly, we could not find this bug with our fuzzer alone. However, we were able to find it as soon as we

used the certificate corpora mutator (see Section 10.3). The certificates used in our certificate corpora came from the OpenSSL corpus itself, which shows that it sometimes is not enough to check the isolated functions, like the certificate parsing but the complete product. Altogether we found eight different cases of undefined behavior.

10.13. Conclusion

We presented a protocol-aware hybrid greybox fuzzer for TLS libraries. We showed that it outperforms current protocol-aware fuzzers (like TLS-Attacker or tlsbunny) and protocol-unaware fuzzers like AFLnet in terms of code coverage. Additionally, we showed that our approach performs better (in almost all cases) than the current best-handwritten test suites (i.e., tlsfuzzer). ETF uncovered 25 previously unknown issues in TLS client implementations. To the best of our knowledge, the hybrid approach used in ETF is currently the best technique to fuzz complex cryptographic protocols like TLS, especially when the application under test can not be modified to disable cryptographic checks. However, the development cost of tools like ETF for other protocols should not be underestimated, especially if specialized libraries like TLS-Attacker are not available yet. The proposed hybrid technique may be a good choice for protocols with many different implementations or very high-security demands.

One of the major shortcomings of ETF is that the analyzer does not understand the semantics behind the corrupted messages ETF generates. If ETF had an objectively correct view of the exchanged message flows, the analyzer could find more complex semantic issues. Although ETF has a lower number of executions per second than AFLnet, it could still reach more code coverage in significantly less time. As such, any future work on improving the speed of ETF will likely be reflected in its fuzzing performance. Other fuzzing techniques like snapshot fuzzing [17, 184] could also help ETF to execute faster, especially since ETF is currently not taking advantage of advances in process restarting mechanisms.

Another interesting research direction could cover other architectures and compiler flags. TLS implementations are primarily configured with compile-time flags that enable or disable entire features or switch the code flow to different implementations (e.g., with different CPU/memory tradeoffs). Security researchers do not as commonly analyze these non-default configurations, making them an excellent target for future research.

11

State Machine Learning

The TLS protocol is a complex cryptographic protocol that requires a non-trivial state machine to implement. Implementations must implement the state machine correctly; otherwise, the security of TLS is at risk. Doing so is challenging, as the TLS RFCs do not explicitly define how the state machine should be implemented, but instead only implicitly describe when which message should be processed and how an implementation should respond. One notable exception is RFC 8446 [53] (TLS 1.3), where the RFC displays an incomplete state machine sketch. State machine learning (also known as state machine fuzzing) is a technique to automatically infer a state machine description of a protocol implementation using *model learning* [185, 186]. It is an automated black-box technique that sends selected sequences of messages to the implementation, observes the corresponding outputs, and produces a Mealy machine that abstractly describes how the implementation responds to message flows. The Mealy machine can then be analyzed to spot flaws in the implementation's control logic or check compliance with its specification. State machine learning works without any *a priori* knowledge of the protocol state machine, but relies on a manually constructed protocol-specific test harness, also known as a *mapper*, which translates symbols in the Mealy machine to protocol packets exchanged with the implementation.

For this work, I implemented the state machine learning approach with TLS-Attacker as a mapper. With my assistance, Paul Fiterau Broştean then adapted this state machine learner to DTLS based on an unfinished rewrite of DTLS for TLS-Attacker that I developed. The experiments with the DTLS implementations were conducted by Paul, while I assisted him with the analysis of the extracted state machines and the discovered vulnerabilities. Paul was further supervised and assisted by Bengt Jonsson and Konstantinos Sagonas, while I was assisted by Juraj Somorovsky. This cooperation resulted in the publication at the USENIX Security Conference 20' [20] which is presented here.

11.1. Background on Model Learning

Mealy machines are finite state automata with finite alphabets of input and output symbols. They are widely used to model the behavior of protocol entities (e.g., [187, 188]). Starting from an initial state, they process one input symbol at a time. Each input symbol triggers the generation of an output symbol and brings the machine to a new state.

We use model learning to infer a Mealy machine model of an implementation. An analyzed implementation is referred to as the system under test. Model learning is an automated black-box technique that *a priori* needs to know only the input and output alphabets of the SUT. The most well-known model learning algorithm is Angluin's L^* algorithm [189], which has been refined into more efficient versions, such as the TTT algorithm [190]. These algorithms assume that the SUT exhibits deterministic behavior and produce a deterministic Mealy machine.

Model learning algorithms operate in two alternating phases: hypothesis construction and hypothesis validation. During hypothesis construction, selected sequences of input symbols are sent to the system under test, observing which sequences of output symbols are generated in response. The selection of input sequences depends on the observed responses to previous sequences. When certain convergence criteria are satisfied, the learning algorithm constructs a *hypothesis*, which is a minimal deterministic Mealy machine consistent with the observations recorded so far. This means that for input sequences sent to the SUT, the hypothesis produces the same output as the one observed from the SUT. The hypothesis predicts an output for other input sequences by extrapolating from the recorded observations. To validate that these predictions agree with the behavior of the SUT, learning then moves to the validation phase, in which the SUT is subject to a conformance testing algorithm that aims to validate that the behavior of the SUT agrees with the hypothesis. Suppose conformance testing finds a counterexample, i.e., an input sequence on which the SUT and the hypothesis disagree. In that case, the hypothesis construction phase is reentered to build a more refined hypothesis that also takes the discovered counterexample into account. If no counterexample is found, learning terminates and returns the current hypothesis. This is not an absolute guarantee that the SUT conforms to the hypothesis, although many conformance testing algorithms provide such guarantees under some technical assumptions. If the cycle of hypothesis construction and validation does not terminate, this indicates that the behavior of the SUT cannot be captured by a finite Mealy machine whose size and complexity is within reach of the employed learning algorithm.

Model learning algorithms work in practice with finite input alphabets of modest sizes. In order to learn realistic system under test's, the learning setup is extended with a so-called mapper, which acts as a test harness that transforms input symbols from the finite alphabet known to the learning algorithm to actual protocol messages sent to the SUT, as illustrated in Figure 11.1. Typically, the input alphabet consists of different messages, often refined to represent interesting variations, e.g., concerning the key exchange algorithm. The mapper

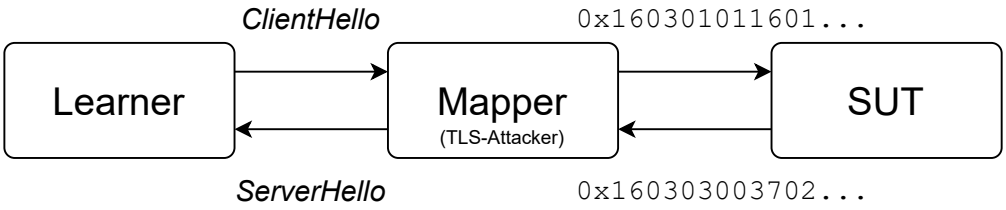


Figure 11.1.: DTLS Learning Setup

transforms each such symbol into a message by supplying message parameters, performing cryptographic operations, and encodes the message. Conversely, the mapper translates output from the system under test into the alphabet of output symbols known to the learning algorithm. The mapper also maintains state that is hidden from the learning algorithm but needed for supplying message parameters, i.e., sequence numbers or agreed encryption keys. The choice of input alphabet and the design of the mapper require domain-specific knowledge about the tested protocol. Once the mapper has been implemented, model learning proceeds entirely automatically.

11.2. DTLS State Fuzzing Framework

In order to apply state machine fuzzing to (D)TLS, we implemented our own testing framework. The learning setup[1] comprises three components: the learner, the mapper, and the system under test (see Figure 11.1). The system under test is a DTLS server implementation, though our setup can be easily adapted to support clients. The learner generates inputs from a finite alphabet of input symbols. The mapper transforms these inputs messages and *records* and sends them over a datagram connection to the system under test. The mapper then captures the system under test's reply, translates it to symbols in the alphabet of output symbols, and delivers them back to the learner. The learner finally uses the information from the exchanged input and output symbols sequences to generate a Mealy machine, as described in Section 11.1.

The learner is implemented using LearnLib [191], a Java library implementing algorithms for learning automata and Mealy machines. The library also provides state-of-the-art conformance testing algorithms used by the learning algorithm for hypothesis validation.

The learning algorithm chosen is TTT [190], a state-of-the-art algorithm that requires fewer test inputs compared to other algorithms [192]. For conformance testing, we use Wp [193] and a variation of it, Wp-Random [194].

Table 11.1 displays the alphabets of input and output symbols and the shorthands that we use to make their representation more compact. The input alphabet includes only the messages that that the client can send in a benign connection. Additionally, it includes `Application` for sending a simple application message and two common alert messages (`CLOSE_NOTIFY` and

[1] `https://github.com/assist-project/dtls-fuzzer/`

Symbol	Shorthand
`ClientHello`(DH,ECDH,RSA,PSK)	CHT
`CertificateRequest`	CertReq
`ClientKeyExchange`(DH,ECDH,RSA,PSK)	CKET
`CertificateVerify`	CV
`Certificate`(RSA, ECDSA, Empty)	Cert(T)
`ChangeCipherSpec`	CCS
`Application`	App
Alert(Close Notify)	A(CN)
Alert(UnexpectedMessage)	A(UM)
Alert(BadCertificate)	A(BC)
Alert(DecodeError)	A(DE)
Alert(DecryptionError)	A(DyE)
Alert(InternalError)	A(IE)
`HelloVerifyRequest`	HVR
`ServerHello`	SH
`ServerHelloDone`	SHD
`ServerKeyExchange`(DH,ECDH,PSK)	SKE(T)
`Finished`	FIN
No Response	-
Disabled	Disabled
Unknown Message	UM

Table 11.1.: Symbols used during learning and their shorthands. We list only the output symbols which are mentioned in the paper.

`UNEXPECTED_MESSAGE`). Interpretations for the alerts can be found in the TLS 1.2 specification [52, p. 31]. Finally, `Certificate`, `Empty Certicicate`, and `CertificateVerify` are included for sending certificate-related messages. `Certificate` contains a single valid certificate and is parameterized by the public key signing algorithm. `Empty Certicicate` denotes sending a certificate message with an empty list of certificates.

The output alphabet includes abstractions for each different message the system under test responds with, similarly to the input alphabet. It also includes three special outputs: '-', when the system under test does not respond; `Disabled`, when the system under test process is no longer running; and `UM`, when the system under test responds with a message which the mapper cannot decrypt. The `UM` symbol can, for example, appear if the mapper has replaced the keys necessary to decrypt the output by a new set of keys.

11.3. Mapper

The mapper uses TLS-Attacker to translate between learner inputs/outputs and actual DTLS messages. Behaviorally, the mapper operates like a DTLS client, with control flow deferred to the learner. In order to reduce the learning effort, we do not subject the SUT to message reordering or fragmentation. Hence, the mapper is configured to send each handshake message in one single DTLS fragment. To correctly supply and check DTLS-specific fields in messages, the

mapper maintains the state of the interaction in a TLS-Attacker `Context`, which it uses to generate and parse messages. The behavior of the mapper is crucial for learning and shall be briefly described. Key components of the `Context` are cookie, cipher state, and digest, as well as the message sequence numbers. Each message sent is equipped with the value of the sequence number, which is then incremented. The mapper also maintains analogous state variables for record sequence numbers and epochs that are incremented whenever a `ChangeCipherSpec` is sent. These variables are also used to assemble fragments into messages and detect retransmissions. Retransmissions here refer to messages whose message sequence number or epoch are smaller than expected. The variable *cookie*, initially set to empty, retains the value of the cookie field in the most recent `HelloVerifyRequest` message received from the server, and is used when sending subsequent `ClientHello` messages. The variable *cipher state* stores the next symmetric keys to be used for decrypting/encrypting messages. To be put in use, a cipher state first has to be *deployed*. The cipher state deployed initially is set to null (no encryption/decryption). On each `ClientKeyExchange` sent, the cipher state is updated using information from an earlier `ClientHello-ServerHello` exchange. On each `ChangeCipherSpec` sent, the cipher state is deployed. This implies that the mapper will only start encrypting/decrypting once `ClientHello` and `ServerHello` are exchanged and a `ClientKeyExchange` and a `ChangeCipherSpec` have been issued. Prior to these actions, messages are sent in plaintext. The variable *digest* stores a buffer of all handshake messages sent so far, i.e., each handshake message that is sent or received is also appended to the *digest*. A hash over this variable is included in every `Finished` message sent, to be verified by the server. The variable *digest* is cleared after each `Finished`, and also before sending `ClientHello`. This strategy for resetting the *digest* enables handshakes to 'restart in the middle', by ensuring that hashes are computed over exactly the messages in the most recent current handshake. After experimenting with different strategies for resetting `digest`, we found that this strategy allows handshakes that restart to complete, whereas other strategies do not. It also produces smaller models since successful restarts typically show up as back-transitions to regular handshake states. For example, for tinydtls using a PSK configuration, the number of states in the learned model was reduced from 36 if `digest` was not reset, to 22 if it was.

11.4. Making the SUT Behavior Deterministic

As mentioned in Section 11.1, the learning algorithm employed works under the assumption that the system under test exhibits deterministic behavior, i.e., the output generated depends uniquely on the supplied input sequence. During learning experiments, however, timing effects occasionally manifest as non-determinism to the time-agnostic learner. Below, we describe our strategies to remedy this problem.

One cause for timing-induced non-determinism is the learner sending the first input too early, before the system under test has fully started, or the mapper determining prematurely that the system under test does not respond. We

address this by tailoring, the start and response timeouts for each SUT. These are, respectively, the delay before the first input is sent (allowing the system under test to initialize), and the time the mapper waits for each response before concluding a timeout. In order to reduce learning time, we adjust the response timeout for certain messages, particularly `ClientHello` and `Finished`, to which the system under test could take longer to respond. Finally, to optimize the start timeout for the slower JSSE and Scandium implementations, we wrap around the system under test a program that preloads key material, among other things. This key material is then reused rather than reloaded for each new sequence of inputs. Once the server is ready to receive packets, the wrapper program notifies the learner of the port number at which the server is listening. The learner can then immediately start sending inputs rather than waiting for a predefined period.

Another cause for non-determinism is timeout-triggered retransmissions by the SUT. To address this, we set the retransmission timeout of the system under test to a high value. For some system under tests, this is a configurable parameter; for others, we had to alter the source code. Corresponding patches are provided on the learning setup's website for reproducibility. Even with the above strategies, an SUT would sometimes produce alternative outputs due to spurious timing effects. In order to detect such cases, we store system under test's responses to queries in a cache during the hypothesis construction phase and confirm each counterexample produced by hypothesis validation before delivering it to the learner. When detecting a case of differing responses to the same input, we rerun the sequence until at least 80% of the responses are the same; this always happened within a small number of retrials.

11.5. Experimental Setup

In total, we analyzed 13 different implementations. This includes well-known TLS implementations like OpenSSL, GnuTLS, mbedTLS, JSSE, wolfSSL, and NSS, which also support DTLS. For JSSE, we analyzed the Sun JSSE provider of Java 9 and 12. Furthermore, we analyzed Pion DTLS, a Go implementation of DTLS 1.2 for WebRTC. The remaining implementations are IoT-specific and support only DTLS. Scandium is the DTLS implementation from Eclipse's CoAP implementation 'Californium' and is written in Java. The two tinydtls variants are lightweight implementations specifically designed for IoT devices. tinydtls for Contiki-NG branched out from that in Eclipse's IoT suite and has been developed independently ever since. We refer to Eclipse's variant as tinydtls^E, and to Contiki-NG's as tinydtls^C. When referring to both, we use tinydtls. For GnuTLS and Scandium, we analyzed two versions; the later version contains bug fixes uncovered in the earlier one. As with tinydtls, we omit versions when referring to both.

To avoid having to write our own DTLS servers, we use utilities to configure and launch DTLS servers that are provided by the developers where possible. For example, for OpenSSL, we use the `s_server` utility, for GnuTLS we use `gnutls-serv`, etc. There are three exceptions (Pion DTLS, Scandium, and

Name	Version	Utility	Algorithms	Client Cert Auth	URL
GnuTLS	3.5.19	`gnutls-serv`	DH,ECDH,RSA,PSK	NONE,REQ,OPT	`https://www.gnutls.org`
	3.6.7		DH,ECDH,RSA,PSK	NONE,REQ,OPT	
JSSE	9.0.4	-	DH,ECDH,RSA	NONE,REQ,OPT	`https://www.oracle.com/java/`
	12.0.2		DH,ECDH,RSA	NONE,REQ,OPT	
mbedTLS	2.16.1	`ssl-server2`	DH,ECDH,RSA,PSK	NONE,REQ,OPT	`https://tls.mbed.org`
NSS	3.46	`tstclnt`	DH,ECDH,RSA	NONE,REQ,OPT	`https://nss-crypto.org`
OpenSSL	1.1.1b	`openssl s_server`	DH,ECDH,RSA,PSK	NONE,REQ,OPT	`https://www.openssl.org`
PionDTLS	e4481fc	-	ECDH,PSK	NONE,REQ,OPT	`https://github.com/pion/dtls`
Scandium^{old}	c7895c6	-	ECDH,PSK	NONE,REQ,OPT	`https://www.eclipse.org/californium/`
Scandium^{new}	6979a09		ECDH,PSK	NONE,REQ,OPT	
tinydtls^{C}	53a0d97	`dtls-server`	ECDH,PSK	*NONE*,REQ	`https://github.com/contiki-ng/tinydtls`
tinydtls^{E}	8414f8a	`dtls-server`	ECDH,PSK	*NONE*,REQ	`https://github.com/eclipse/tinydtls`
wolfSSL	4.0.0	`server`	DH,ECDH,RSA,PSK	NONE,REQ,OPT	`https://www.wolfssl.com`

Table 11.2.: Overview of tested DTLS implementations. '-' means a custom program was provided. Client certificate authentication can be disabled (NONE), required (REQ), and optional (OPT). Grayed out or slanted are configurations supported by the library but not made available by the utility. For slanted configurations, this support was added. Braces gather configurations explored via single learning experiments.

JSSE) for which we wrote our own DTLS applications[2] as there were no standard utilities available or the available ones did not provide the desired functionality.

For every implementation, Table 11.2 displays the name, version, utility, supported key exchange algorithms and client certificate authentication configurations, and a reference. We use commit identifiers as versions for both tinydtls variants, Pion DTLS, and Scandium. The two commits for Scandium belong to the development version 2.0.0 and shall be referred to as Scandium^{old} and Scandium^{new}. Note that client certificate authentication is relevant for DH, ECDH and RSA, but not for PSK whose handshake does not incorporate certificate messages [105, p. 4].

The input alphabet, described in Table 11.1, includes inputs necessary to perform handshakes using every key exchange algorithm supported, two alerts, and one application message. Whenever certificates can be part of the key exchange algorithm, they are also included in the alphabet. The system under test is configured to use client certificates whenever these are supported. Therein we explore three configurations:

1. *required*: a valid certificate is requested (via `CertificateRequest` message) and required to complete a handshake;
2. *optional*: a valid certificate is requested but not required; and
3. *disabled*: a valid certificate is neither requested nor required.

These configurations are further detailed in Subsection 11.6.1.

In some experiments, we had to remove inputs from the input alphabet and sometimes limit the set of explored configurations. For Pion DTLS, NSS and

[2]These implementations are accessible via the learning setup's website.

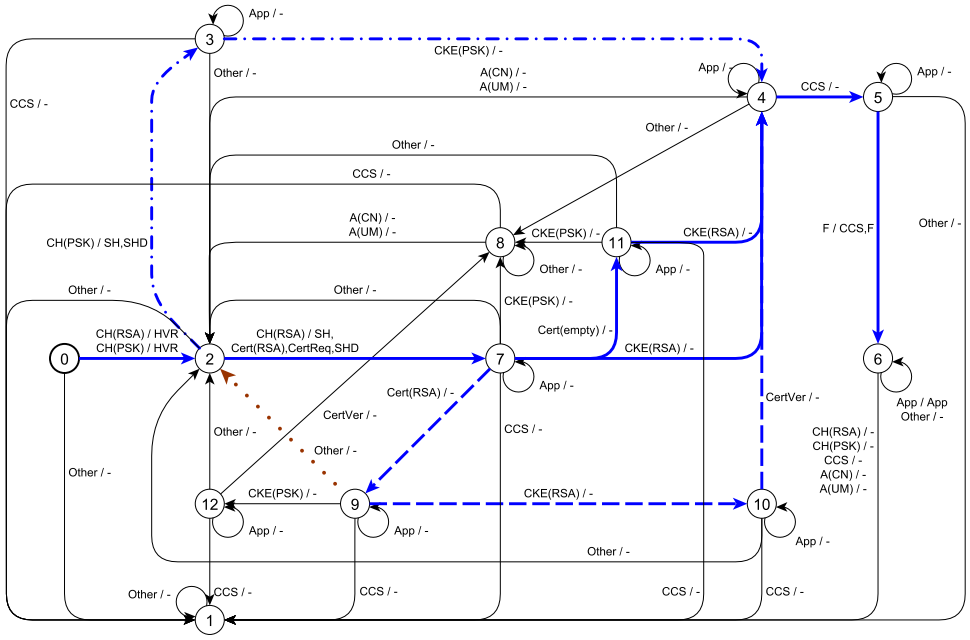


Figure 11.2.: Model of a GnuTLS 3.6.7 server with optional client certificate authentication. Blue edges capture the flows of regular handshakes: dashed and dashed-dotted edges indicate the handshake expected when client certificate authentication is required, respectively, when it is disabled. A dotted brown edge indicates a transition leading to a handshake restart.

wolfSSL, the reason was that the server program or library does not support certain combinations of key exchange algorithms and certificate configurations. Similarly, Pion DTLS's library does not allow PSK and ECDH cipher suites to be used together, NSS's utility does not support certificate authentication, while wolfSSL's utility could not be configured to support all key exchange algorithms simultaneously. In cases where learned models were large (for tinydtls, Scandium, and JSSE) or when response time was slow (for Scandium and JSSE), we generated models separately for each key exchange algorithm in order to keep the learning time reasonable.

11.6. Evaluation

This section provides an analysis of the models in regard to the DTLS specification. We first give an overview of a DTLS state machine, using the model learned for GnuTLS as an example. We explain the strategies employed to identify non-compliant behaviors using the learned models. We then outline the non-compliant behaviors observed in the tested libraries. Finally, we present library-specific findings and vulnerabilities, including the client authentication bypass in JSSE.

11.6.1. Reading State Machine Models

Displaying models is challenging due to the large number of inputs and states. We, therefore, prune the models via the following strategies. We first use the *other* input as a replacement for inputs not captured in a visible transition which lead to the same state and output. Inputs and outputs are then replaced by their corresponding shorthands shown in Table 11.1. Finally, we place transitions connecting the same states on single edges. The models for GnuTLS 3.6.7, JSSE 12.0.2, and Pion DTLS are included in this dissertation. All other models can be accessed via the learning setup's website[3].

Figure 11.2 shows a model generated for the GnuTLS 3.6.7 library and can be interpreted as follows. The server starts from the initial state, which is always state 0 on the state machine. On receiving `ClientHello` it generates `HelloVerifyRequest` and transitions to state 2. In response to a second `ClientHello`, it generates the messages `ServerHello` and `ServerHelloDone` and transitions to state 3. Continuing the PSK handshake flow, on receiving `ClientKeyExchange`, `ChangeCipherSpec` and `Finished`, the server does not respond to the first two messages and responds with a `ChangeCipherSpec` and `Finished` to the third. In this interaction, the server traverses the states 4 and 5, ending in 6.

The GnuTLS server was configured to use PSK- and RSA-based cipher suites. This is reflected in the model's input alphabet, which includes `ClientHello` and `ClientKeyExchange` for both PSK and RSA. Client certificate authentication was set to optional. In this situation, the server makes a client certificate request, as indicated by the `CertificateRequest` label on the edge from state 2 to state 7 in Figure 11.2. The server does not require client certificates; hence handshakes can be completed even if the client chooses to send an empty `Certificate`. This can be achieved by following states 0, 2, 7, 11, 4, 5, and 6; or without a `Certificate` message at all by following states 0, 2, 7, 4, 5, and 6. Finally, if the client authenticates with a `Certificate` message, the handshake traverses states 0, 2, 7, 9, 10, 4, 5, and 6. Note that client certificate authentication is implicitly disabled for cipher suites that do not support it, such as PSK-based ones.

Besides states traversed by handshake flows, the model contains three other states: states 1, 8, and 12. State 1 is a *sink state*, which is a state that cannot be left once entered. States 8 and 12 are *superfluous states*, since they are not necessary for implementation correctness. They are a byproduct of the implementation allowing handshake restarts, which are possible from these states by transitions to state 2.

11.6.2. Identifying Irregular Behaviors

We employ the following strategies to identify potentially vulnerable behaviors using learned models.

First, we inspect models for irregular handshake flows (*irregular handshakes* for short). These are flows that lead to handshake completion, indicated by a successfully transmitted `Finished` from the server, but may omit, repeat or

[3] `https://github.com/assist-project/dtls-fuzzer/tree/master/experiments`

change the order of handshake messages relative to regular flows permitted by the specification. To aid analysis of larger models (such as those of JSSE or Pion DTLS) we developed a script to automatically remove states from which a handshake cannot be completed (i.e., it is no longer possible to receive a `Finished` from the server). On the reduced models, handshake-completing flows can be identified much more easily; this is showcased by Figure 11.3 and Figure 11.4. Using this approach, we uncovered bugs wherein a handshake is completed by omitting the `ChangeCipherSpec` message. We refer to Sections 11.6.4 ,11.6.5 and 11.6.6 for descriptions of such bugs for JSSE, Scandium and Pion DTLS. Note that the script used to reduce models comes packaged with our learning setup.

Second, we look for outputs from the server which do not conform to the specification. Of particular interest are irregular `ServerHello` responses, which are not part of irregular handshakes (otherwise, the flows would have been detected and analyzed by our first strategy). We investigate whether a handshake may be completed using these responses. To that end, we probe the system under test's reaction after such responses to manually-crafted messages (typically `ClientKeyExchange`, `ChangeCipherSpec` and `Finished`), whose message sequence or epoch numbers differ from what our mapper generates. Doing so, we could complete handshakes in tinydtls using invalid epoch numbers; see Subsection 11.6.8. Also of interest are `Alert` outputs, as they shed light on how the system processes unexpected inputs. For example, alert `DECRYPTION_ERROR` suggests the system under test is not able to decrypt a message. Hence, `DECRYPTION_ERROR` is only expected as a response to an encrypted message and not to an unencrypted message, as was the case for tinydtls; see Subsection 11.6.8.

Finally, we inspect the code exercised by irregular behaviors identified by the first two strategies in order to assess whether they can result in further flaws. Such flaws can be more severe than the initial irregularity suggests. As an example, the non-conforming `DECRYPTION_ERROR` in tinydtls led us to discover loss of reliability in the face of reordering. Investigation can also reveal bugs not directly related to the behavior inspected, which, however, exercise roughly the same portion of code. Such was the case for Pion DTLS, where investigating an early `Finished` bug led to the discovery of premature processing of application data; see Subsection 11.6.6.

11.6.3. General Behavior Patterns

Several conforming and non-conforming behavior patterns emerged while analyzing the learned models. Table 11.3 summarizes the irregular behaviors and the affected implementations.

Handshake with Invalid Message SQN. Many DTLS server implementations allow for creating new associations even with an already established connection [7, Section 4.2.8]. This process involves performing a new `ClientHello`–`ServerHello` exchange in the middle of an already started or finished handshake and results in agreeing on a new cipher suite and key material. The motivation behind this behavior is to support clients that want to re-establish a new

Library	Validation of message SQN	Cookie computation	Message order verification
GnuTLS	✗	✗	✓
JSSE_{old}	✓	✓	✓
JSSE_{new}	✓	✓	✗
mbedTLS	✗	✗	✓
NSS	✓	✗	✓
OpenSSL	✓	✗	✓
Pion DTLS	✓	✓	✗
Scandium_{old}	✗	✓	✗
Scandium_{new}	✗	✓	✓
tinydtls	✗	✓	✓
wolfSSL	✗	✓	✓

Table 11.3.: Summary of irregular behaviors detected in the tested libraries. The first column summarizes the correct usage of these numbers. Yindicates that the implementation finished the handshake with an invalid *message SQN*. The second column summarizes the cookie computation correctness. The last column depicts whether implementations correctly validate the handshake message sequence.

connection after losing one (e.g., after a reboot). According to the DTLS specification [7, Section 4.2.2], every `ClientHello` starting a new handshake must have *message SQN* = 0. Every following handshake message has to increase the *message SQN* by one.[4]

In five of the tested implementations, starting a DTLS handshake with a higher *message SQN* was possible. It was also possible to identify these implementations from the learned models. For example, in the GnuTLS model (Figure 11.2), discovered such an invalid behavior by following the transitions looping back to state 2.

Non-conforming Cookie Computation. Upon receiving a `ClientHello` message, the server computes a stateless cookie and sends it via `HelloVerifyRequest`. The server expects the cookie to be replayed in the subsequent `ClientHello` message. According to the specification, the replayed `ClientHello` message must contain the same parameters as the first one (e.g., supported cipher suites) [7, Section 4.2.1]. For this purpose, the server should use the initial `ClientHello` parameters to compute the cookie value. In our evaluation, we could observe four implementations incorrectly computing the cookie value, resulting in incorrect validation of replayed `ClientHello` messages. Such a handshake is also captured in Figure 11.2, where an RSA handshake can be completed even if the first message was `ClientHello`. An exceptional case

[4] As mentioned in Section 3.18, DTLS also defines explicit sequence numbers in DTLS records. In contrast to *message SQN* located in handshake messages, an implementation can accept a DTLS record with a SQN that was increased by more than one.

is NSS, which omits the cookie exchange step altogether, in discord with the specification's recommendation.

Handshake with Invalid Order of Messages. The most consequential divergent behaviors are handshakes, where invalid message sequences lead to handshake completion. These behaviors may have severe security implications. We found that JSSE, Pion DTLS, and Scandium$_{old}$ do not correctly verify the DTLS handshake message sequence in their internal state machines. Below we discuss these bugs and their implications.

11.6.4. JSSE

Figure 11.3 depicts the hypothesis model generated for JSSE using one RSA-based cipher suite after two days of learning. The model was obtained by erasing all states from which a handshake could no longer be completed. The JSSE server was configured to require client authentication.

The model depicts a correctly completed handshake, which is marked with blue edges and follows states 0, 2, 4, 11, 12, 3, 9, and 10. This flow includes `Certificate` and `CertificateVerify` messages correctly sent by the client to authenticate to the server. However, even though the server required client authentication, we could complete a DTLS handshakes without sending `Certificate` or `CertificateVerify` messages. The invalid handshakes are captured in red and allow a client to bypass client authentication. Our analysis revealed that versions 11, 12, and 13 of Oracle and OpenJDK Java are affected for all key exchange algorithms. Previous versions are not affected by this issue.

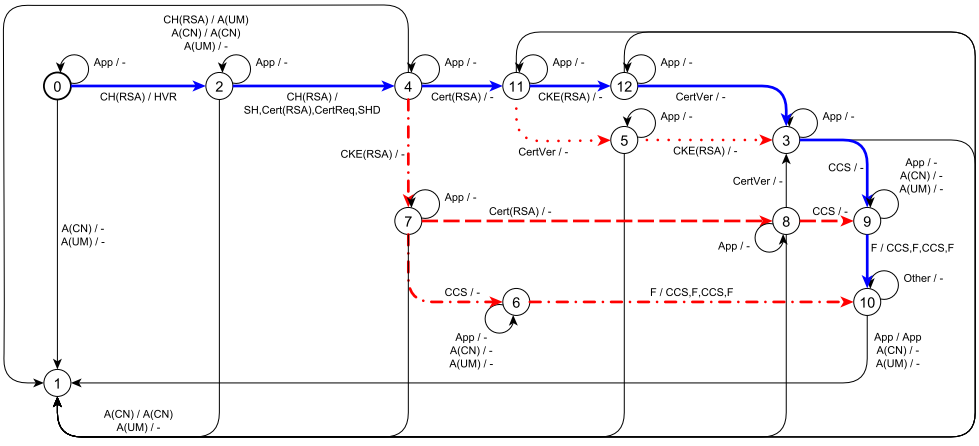


Figure 11.3.: Model of a JSSE 10.0.2 server with client authentication set to required. Blue edges capture the happy flow, dotted red a handshake with an unauthenticated `ClientKeyExchange` message, dashed-dotted red a handshake without certificate messages, dashed red a handshake without `CertificateVerify`.

Unauthenticated ClientKeyExchange. We start the description of JSSE vulnerabilities with a slightly modified happy flow, which follows states 0, 2, 4, 11, 5, 3, 9, and 10, and traverses dotted red edges on the model. In this flow, the client sends a `CertificateVerify` message before the `ClientKeyExchange`. This implies that the `ClientKeyExchange` message is not authenticated with the client certificate.

Finalizing such a DTLS handshake does not directly result in a critical vulnerability. Suppose the client behaves correctly and sends messages in the correct order. In that case, an attacker cannot modify the `ClientKeyExchange` message or the message order because all the handshake messages are protected by the `Finished` message. Still, this bug shows a first invalid behavior and scratches the surface of other invalid ones.

Certificate-less Client Authentication. The second vulnerability is marked with dashed-dotted red edges in Figure 11.3. The DTLS handshake starts with four ordinary flights of messages. The server requests client authentication in the fourth flight by sending a `Certificate` message. However, the client ignores this message and continues the handshake with `ClientKeyExchange`, `ChangeCipherSpec`, and `Finished` messages, without sending `Certificate` and `CertificateVerify`. The server responds to the last message with `ChangeCipherSpec` and `Finished`, thus completing the handshake. This allows the client to completely bypass client authentication and proceed with sending application data.

Note that the handshake process remains entirely transparent for the server, as long as the server does not try to inspect the peer's certificate after completing the handshake manually. Since the client does not send any certificate, the certificate in the internal JSSE context is *null*. If the server attempts to evaluate the certificate data (e.g., to access the subject name or certificate issuer fields), this will result in an `SSLPeerUnverifiedException` and most likely interrupt the authentication process. The following finding bypasses this constraint as well.

CertificateVerify-less Client Authentications. The third vulnerability follows red dashed edges in Figure 11.3 and partially relies on the above behavior. It allows an attacker to authenticate as an arbitrary user without possessing the private key. The only prerequisite is that the attacker is in possession of a valid client certificate. This requirement is usually trivially achieved as certificates are usually not considered private and can be found in public repositories or provided in frameworks like Certificate Transparency.

As already visualized on the model, after receiving the second server message flight, the attacker can send a `ClientKeyExchange` message, thus transitioning from 4 to 7. Instead of directly sending a `ChangeCipherSpec` message, we continue with an out-of-order `Certificate` message. Finally, we send `ChangeCipherSpec` and `Finished`. The server then responds with `ChangeCipherSpec` and `Finished`, after which it can accept an `Application` message encrypted under the established keys. Thus, the attacker can finalize the DTLS handshake

without `CertificateVerify`, and thus without possessing the certificate's private key. The crucial difference compared to the previous vulnerability is that the server accepts the certificate and can process its contents correctly. Therefore, no `SSLPeerUnverifiedException` is thrown, and the application cannot detect invalid client behavior.

Attack Rationale and State Machine Analysis. To understand the observed behaviors, we analyzed the JSSE state machine implementation. The reason behind the vulnerabilities is not intuitive. In general, it can be summarized in the following processing properties.

- First, the server does not validate a proper message order. From the first bug, we can conclude that specific handshake messages can be sent in a different order (e.g., `ClientKeyExchange` and `CertificateVerify`).
- Second, the server only partially validates the correctness of received messages. For example, it validates whether the handshake contains a `ClientKeyExchange` message, or it does not accept further `ClientHello` messages after a `ServerHelloDone` message has been sent.
- Third, and most importantly, the server does not verify the presence of critical messages after the handshake has been finalized. In particular, it does not check whether `Certificate` and `CertificateVerify` messages were received after a `Certificate` has been sent.

Our code analysis revealed that the JSSE implementation always waits for at least `ClientKeyExchange`, `ChangeCipherSpec`, and `Finished` messages. Messages arriving out-of-order can be cached. This explains why we could observe so many different paths leading to handshake completion in the learned model.

Interestingly, the bugs similarly affect the TLS implementation as well. Omitting the `Certificate` and `CertificateVerify` messages also authenticates the client. Additionally, just removing the `CertificateVerify` message (while leaving the `Certificate` message) also authenticates the client. We were able to reproduce the issues with Apache Tomcat 9.0.22, which was configured with JSSE and required client authentication.[5] We reported the vulnerabilities to the Oracle security team. They were assigned CVE-2020-2655 and patched with the Oracle critical patch update in January 2020.

11.6.5. Scandium

Scandium$_{old}$ produced some of the largest models. This is reflective of the fact that the implementation did not use an internal state machine to validate the sequence of handshake messages. Consequently, its model captures handshakes with invalid sequences of messages. Reporting our findings prompted Scandium developers to update the implementation with state machine validation

[5] It is also possible to configure Apache Tomcat with an OpenSSL engine (`https://tomcat.apache.org/tomcat-9.0-doc/ssl-howto.html`). This version was not affected.

(Scandium$_{new}$). This update fixed all the Scandium bugs reported in this paper. The update helped simplify the learned model (for a PSK configuration reducing the size from 16 to 13) and enabled convergence for ECDH configurations resulting in similarly small models. Models for the original and updated versions are available online. Below, we present findings for the original version.

Early Finished. Scandium allows a handshake to be completed without the client sending a `ChangeCipherSpec` message. The server then interprets all the upcoming messages as sent in plaintext. It still expects a valid `Finished` message with correct `verify_data` from the client to complete the handshake. Therefore, a MitM attacker is not able to simply drop `ChangeCipherSpec` and use a fabricated `Finished` message to decrypt the traffic. A valid `verify_data` would still be required to complete the handshake. This is impossible without possessing the master secret or exploiting other bugs. However, this behavior shows the fragility of the Scandium state machine.

The early `Finished` message bug is remarkably similar to the bug reported for JSSE 1.8.0_25 [195], and is related to the attack described by Wagner and Schneier [196]. An attacker could exploit this behavior by injecting a backdoor into a library, forcing a DTLS client to skip `ChangeCipherSpec` messages. The attacker could then observe plaintext connections established with any Scandium server.

Multiple CCS in a Handshake. Scandium can complete handshakes wherein `ChangeCipherSpec` is followed by one or more `ChangeCipherSpec` messages and then `Finished`. On each `ChangeCipherSpec` sent the mapper increments the epoch used in follow-up messages. Thus, the sent `Finished` carries an epoch number for which a cipher has not been negotiated. The fact that Scandium completes handshakes in such a situation further showcases the looseness of its implementation.

Measurable Improvements. After we reported the vulnerabilities to the Scandium developers, they were able to simplify Scandium's state machine significantly. Scandium$_{new}$ generates at most 17 states, whereas Scandium$_{old}$ generates up to 45 in a more restricted setting.

11.6.6. Pion DTLS

Early `Finished` Revisited. Pion DTLS exhibits an early `Finished` message bug, similar to the one found in Scandium. Obtained for a server requiring certificate authentication, Pion DTLS's model (Figure 11.4) captures three handshakes instead of the one expected. The two additional handshakes are an early `Finished` handshake and a handshake with a `ChangeCipherSpec` message preceding `CertificateVerify` (where the `CertificateVerify` is sent encrypted). This latter bug clearly shows that Pion DTLS does not correctly validate the ordering of messages.

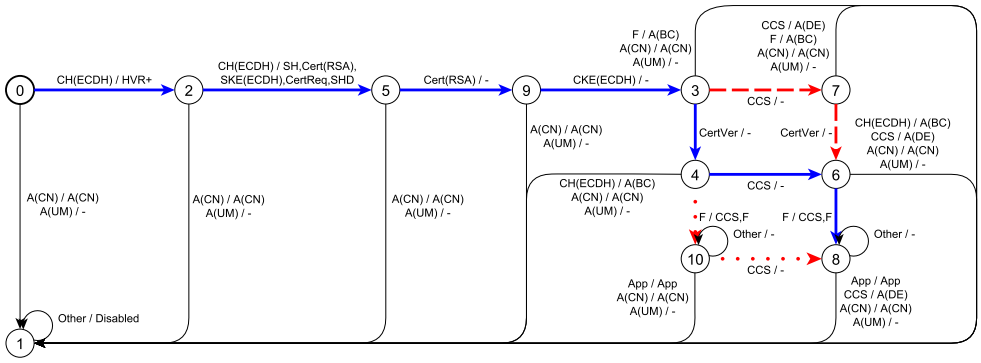


Figure 11.4.: Model of a Pion DTLS server with client authentication set to required. The model was reduced from 66 states to 11 by retaining only states from which a handshake can be finalized. Dotted red indicates an early `Finished` handshake, dashed red a handshake with a delayed `CertificateVerify` message.

Processing of Unencrypted Application Data. During the analysis of the previous bug, we noticed that Pion DTLS freely processed unencrypted application data delivered with epoch 0. This bug has severe consequences by allowing an attacker to inject arbitrary application data at any point once a handshake has been completed. The bug was promptly fixed once we reported our findings to the developers.

HelloVerifyRequest Retransmissions. Pion DTLS occasionally responds to the first `ClientHello` message with multiple `HelloVerifyRequest` messages. This response is marked with `HelloVerifyRequest+` in Figure 11.4. When investigating this behavior, we found that Pion DTLS will retransmit `HelloVerifyRequest` messages until a timeout elapses or it receives the second `ClientHello`. RFC 6347 advises against retransmitting `HelloVerifyRequest` [7, p. 6], as doing so requires the server to keep state, making it susceptible to Denial-of-Service attacks. The retransmission also enables amplification attacks, wherein an attacker sends `ClientHello` messages to the server with the IP address of a victim as the source address. As a result, the server will send its replies to the spoofed source address, thus flooding the victim with `HelloVerifyRequest` messages.

11.6.7. GnuTLS

In GnuTLS_{old}, we detected a bug in the initial state; the implementation treated most messages as if they were `ClientHello`. In doing so, the server responded to them with `HelloVerifyRequest`, and transitioned to the next handshake state.

We reported the bug to the GnuTLS developers, who were able to reproduce and fix the issue.

11.6.8. tinydtls

Insecure Renegotiation. After performing a DTLS handshake with a tinydtls server, we could use the established encrypted connection to perform a renegotiation. The `ClientHello` messages we used did not contain any renegotiation indication extension. Therefore we consider the server vulnerable to the renegotiation attack. The real exploitability of this behavior depends on the application using the tinydtls library.

Crashes on CCS. In addition, we found that in certain states TinyDTLS^E crashes on receiving `ChangeCipherSpec`. For example, it crashed on receiving this message in the initial state. The crashing behavior resulted in a reduction of states compared to TinyDTLS^C since crashing inputs predictably lead to a single sink state. The crash resulted from a segmentation fault resulting from a null address read. This bug is a rediscovery of CVE-2017-7243, which is still unfixed in the master branch of the TinyDTLS^E.

From Inconsistent Alert to Unreliable Handshakes. By analyzing the learned model, we could observe frequent usage of alert `DECRYPTION_ERROR` messages. This alert is sent by tinydtls whenever it tries to decrypt a record (whether it is encrypted or not) and fails to find key material for the epoch in its internal state. This behavior is relatively unproblematic, but tinydtls also invalidates the whole connection in such a case. This can break connections unnecessarily when the `ChangeCipherSpec` and `Finished` messages are received out of order in a regular handshake.

Handshake with Invalid Epoch Numbers. The model for TinyDTLS^C revealed that the server can perform the first two steps of a handshake using `ClientHello` messages with epoch 1 when no cipher for epoch 0 has yet been negotiated. Upon further investigation, we were able to complete the handshake by sending `ClientKeyExchange`, `ChangeCipherSpec` and `Finished` having the same epochs as in a normal handshake (which are 0, 0, and 1, respectively). The handshake is invalid and should not have been possible to complete.

11.6.9. OpenSSL

Finished Treated as Retransmission. After a successful handshake completion, the OpenSSL server treats retransmitted `Finished` messages incorrectly. OpenSSL responds to a newly computed and transmitted `Finished` message by re-sending the last flight (`ChangeCipherSpec`, `Finished`). The `Finished` message received from the server has a different message sequence number and `verify_data` content. An adequate response would have been either to discard this message or to send an alert and possibly terminate the connection.

Internal Error. An `INTERNAL_ERROR` is sent by OpenSSL in response to unexpected `Finished` messages. Internally, OpenSSL is processing the message and trying to compute the `verify_data` for the `Finished` message. However,

due to defensive programming, missing parameters in the session context are discovered, the processing of the message is stopped, and an `INTERNAL_ERROR` alert is returned. An appropriate response should have been an alert indicating that an out-of-order message was received. The `UNEXPECTED_MESSAGE` alert has been designed for such purposes.

11.7. Conclusion

In our analysis, we observed several repeating code patterns, which led to bugs and vulnerabilities. Most importantly, most analyzed implementations do not use proper state machines. While they attempt to verify the handshake protocol flow with simple checks in switch statements, a complete message flow validation is missing. This was, for example, observed by the analysis of the Scandium implementation, which was too liberal regarding the message sequence verification; only other additional checks in the code prevented further security vulnerabilities. One reason for missing state machines could be the fact that the DTLS specification [7] does not give a design for one. We believe that protocol standards should contain such designs and demand that implementations use them.

In the libraries implementing TLS and DTLS, we could observe that the code is re-used in both protocols. This means that similar vulnerabilities in one protocol implementation can influence the other. For example, we found the authentication bypass in JSSE by analyzing the DTLS server implementation. However, our subsequent analysis revealed that the bug is also applicable to TLS. We expect that similar behaviors will be found in the future. Interestingly, Scandium and Pion DTLS include the same early `Finished` message bug found in JSSE TLS in 2015 [195]. While this may be attributed to missing state machine implementation, we believe this bug is closely related to the ambiguity mentioned in RFC 6347 [7].

> As with TLS, the `ChangeCipherSpec` message is not technically a handshake message [...]. This creates a potential ambiguity because the order of the `ChangeCipherSpec` cannot be established unambiguously with respect to the handshake messages in case of message loss.

In DTLS up to version 1.2, this ambiguity has to be resolved by hard-coding the expected `ChangeCipherSpec` message. For DTLS 1.3 [197], the problem has been resolved by removing `ChangeCipherSpec` messages entirely.

12 Related Work

As the presented dissertation touched on different areas of TLS, there also exists a lot of related work to this. This section gives an overview of the most important work in these areas.

Bleichenbacher Attack. A common attack on the TLS protocol are *Bleichenbacher* attacks [12]. The Bleichenbacher attack was discovered by Daniel Bleichenbacher in 1998. Since the original discovery of the attack, variations and improvements of the attack appear from time to time [92, 173, 198]. A cross-protocol attack with SSLv2 was presented in the DROWN attack [93]. Ronen et al. [199] evaluated cache-based side channels how the attack can be parallelized. Böck et al. [89] performed a study about the prevalence of Bleichenbacher side channel by also accounting for TCP-based side channel and shortened message flows. Jager et al. [200] disscussed the applicability of the Bleichenbacher attack to TLS 1.3. Drees et al. [153] explored Bleichenbacher side channel attacks by considering more potential side channel sources and evaluates them with machine learning algorithms.

CBC Padding Oracles. Another common flaw in TLS libraries are padding oracle vulnerabilities. Serge Vaudenay [62] discovered the original attack, and since then variations of the attack appeared. Paterson and AlFardan [117] described a padding oracle vulnerability in DTLS that uses DTLS specific features. The Lucky13 attack [11] presented a padding oracle vulnerability that stems from the specification of TLS. The attack has since then been reevaluated in slightly different contexts [85, 86]. POODLE [63] is a padding oracle vulnerability in SSLv3 that also affected newer clients due to a downgrade behavior of some clients. Some variants of the POODLE attack also affected TLS [80–83].

BEAST. The BEAST attack [25] is a famous attack on TLS 1.0 in CBC mode. The attack introduced the MitB attacker model to TLS. Before the BEAST attack was discovered, Gregory V. Bard [72, 73] presented the same attack without a MitB.

Diffie-Hellman. The Diffie-Hellman key exchange plays a major role for TLS. The Logjam attack is an attack on TLS-DHE when the server supports `EXPORT`. Dorey et al. [201] evaluated the impact and prevalence of backdoored Diffie-Hellman parameters in TLS.

Old Cryptography. Since TLS is already over 25 years old, a lot of now deprecated ciphers and key exchange algorithms were supported, which led to numerous attacks. Bhargavan and Leurent [96] presented an attack on TLS with block ciphers with 64-bit block length called Sweet32. In the same year at NDSS, Bhargavan and Leurent [97] investigated transcript collision attacks. The Freak attack [101] exploits a common flaw in the state machine of TLS clients if the server supports `RSA-EXPORT` cipher suites. The RC4 stream cipher was the target of attacks in the context of TLS by Vanhoef and Piessens [202] and AlFardan et al. [203].

Compression. Side channel attacks on the cipher text size due to compression are a serious threat for TLS. The first attack of this kind was the CRIME attack [76] which exploited the built-in TLS compression. Since then, TLS compression has been mostly disabled. The Breach attack [78] exploited HTTP compression, while the HEIST [79] and TIME [77] attack exploit attack variants without a MitM.

Renegotiation Attack. Some attacks on TLS are only possible because of specific TLS features outside of the basic key exchange. The first attack of this kind is the 'Renegotiation attack' [64] that exploited the renegotiation feature to perform a chosen prefix attack. The vulnerability was fixed with the introduction of the `RenegotiationInfo` extension, but the fix was incomplete, resulting in the 'Triple Handshake attack' [65].

Implementation Problems. While some of the already presented attacks are present in TLS due to flaws in the specification (Lucky13 [11], Renegotiation attack [64], ...), others are present due to implementation flaws (Bleichenbacher [12], Freak [101], ...). Böck et al. [88] explored the prevalence and exploitability of GCM IV reuse in the context of TLS. The security of elliptic curve implementations in the context of TLS in regards to invalid curve attacks [99] was investigated by Jager et al [100]. The BERserk [35] attack exploited a flaw in the ASN.1 library of NSS to perform the Bleichenbacher06 [34] attack. The Heartbleed [152] bug is a buffer-overread vulnerability in OpenSSL that could be used to extract the private key of affected hosts. Poddebniak et al. [204] investigated implementation defects of STARTTLS. Checkoway et al. [205] analyzed the exploitability of the Dual EC backdoor in the context of TLS.

X.509. The PKI and the X.509 certificate infrastructure are research topics on their own. Attacks on the certificate validation are often times very severe for TLS. A case study on certificate validation code was done by Georgiev et

al [206]. Automatic tools to find flaws in the verification logic were developed by Brubaker et al [166], Sivakorn et al. [170], as well as Chen and Su [207].

Human Factor. TLS is generally not well understood among end users. These interact with the protocol when they see warning messages related to TLS in the browser or have to judge the security implications of the lock in the browser. This human factor was, for example, evaluated by Sunshine et al [208].

Timing Attacks. Side-channel attacks based on timing information are a major concern for TLS implementations. Besides the already mentioned side-channel attacks like Bleichenbacher, Lucky13, or CBC padding oracles, side-channel attacks on signature schemes are also relevant. One of the first attacks of this kind were presented by Brumley and Boneh [209] who were able to extract the private key from the server. The attack was later improved by Aciiçmez et al. [210]. A similar attack against ECDSA was shown by Brumley and Tuveri [211].

Theroy, Proofs, and Verification. Since its creation, the SSL/TLS protocol has been the subject of security analysis. Research teams have been trying to prove different aspects of the TLS protocol secure for some time. Sometimes these approaches revealed new flaws in the protocol which often resulted in real attacks on the protocol down the line. One of the first security analyses was performed by Wagner and Schneier [196]. Further analysis on SSLv3.0 was done by Song and Quin [212], Mitchell et al. [213], and Cheng et al. [214].

TLS-DHE and TLS-DH have been proven secure under the PRF-ODH assumption by Jager et al. [215] and Krawczyk et al. [216], respectively. The Raccoon attack has shown that this assumption is not met by current real-world TLS implementations. TLS-RSA was analyzed by Jonsson and Kaliski [217]. The security of EAP-TLS and TLS exporter secrets was analyzed by Brzuska et al [218]. Paterson et al. [219] analyzed the impact of the truncated HMAC extension [67] on TLS. Pre-shared Key cipher suites have been analyzed by Li et. al [220]. The renegotiation feature was analyzed in-depth by Giesen et al [221]. An overall, modular analysis approach to the TLS protocol was done by Morrissey et al [222].

Besides these more theoretical approaches to the protocol, there also exist verification approaches, that are trying to create a TLS implementation that is formally verified. Research in this direction can be found in Ramananandro et al. [223] and Bhargavan et al [224].

During the development of TLS 1.3, the scientific community was shepherding the whole process, with proofs, verified implementations, and other analyses within different security models which resulted in a huge amount of scientific publications [69, 225–238].

Extending TLS. There exist a lot of scientific papers that try to extend the TLS protocol to achieve more or different security goals. These proposals often

make significant changes to the protocol, like adding additional parties or completely restructuring the handshake, and are highly unlikely to ever be adopted as a standard. There are a few exceptions to this rule. One of these exceptions is Aviram et al. [239], which upgrades the session ticket mechanism in TLS 1.3 to a perfect forward secure session resumption by using puncturable encryption. In contrast to other extensions, this extension only requires server-side changes that are completely transparent to the client and are therefore very easy to integrate.

Protocol Testing. Compared to other software, fuzzing cryptographic protocols is complicated because some messages or parts of messages are encrypted and signed. This makes it impossible to find valid inputs by random search.

Beurdouche et al. [137] created FLEXTLS based on the MITLS library to provide a framework for testing TLS libraries .

Bozic et al. [240] created an LNT model of the TLS 1.3 handshake and applied conformance testing to it. This approach tries to bridge the gap between specification and implementation. A similar combinatorial approach was used by Simos et al. [241] to generate test cases for TLS handshakes. Walz and Sikora [242] applied differential testing to TLS libraries but limit their approach to the `ClientHello` message and focus on the message parser.

Tsankov et al. [243] built a fuzzer for security protocols called SecFuzz and applied it to the IKE protocol. They provide a generic method for fuzzing security protocols but adopting it to TLS would require considerable effort. The method is generic because it mutates the communication between existing implementations and thus, in the case of TLS, would not require implementations of all features. This advantage is unfortunately diminished by the need to mutate encrypted payloads, which requires at least partial implementation of the protocol and message parsing.

Extensive X.509 security evaluations were already in the scope of previous works. The authors used different methods to identify security issues in certificate validations, for example, differential testing or symbolic execution [166–171].

BoarSSL and the Twrch framework were explicitly developed to test the BearSSL library [244]. It does not contain fuzzing abilities, and all tests must be implemented manually.

tlsfuzzer [179] is (despite its name) a test suite for TLS servers. It implements a series of handcrafted scripts that test an implementation for various flaws. Some of its scripts use fuzzing approaches to select input values. Asadian et al. [245] used symbolic execution to find flaws in DTLS implementations.

This dissertation builds at large on the TLS-Attacker framework published in 2016 [15]. While the first TLS-Attacker version provided only basic fuzzing features, together with tlsfuzzer [179] it could be considered the state-of-the-art for protocol-aware TLS fuzzing. Nevertheless, TLS-Attacker had some drawbacks regarding TLS fuzzing. Most notably, it was only used to evaluate TLS server implementations. It also lacked many TLS features and did not use binary instrumentation. This resulted in TLS-Attacker's inability to perform well during

the code coverage evaluation in Section 10.10.

State Machine Learning. Van Drueten [246] obtained some preliminary results on analyzing DTLS implementations using protocol state fuzzing, from which the work in Chapter 11 branched off. His thesis analyzed OpenSSL and mbedTLS with a limited input alphabet and did not reveal any security vulnerabilities. De Ruiter and Poll [195] used protocol state fuzzing to analyze TLS implementations and found several security bugs. In comparison, the models we learn are significantly larger, due to the complexity in DTLS introduced by UDP, and our inclusion of several key exchange algorithms and certificate settings. Also, as stated before, some of the bugs we found are only possible under particular configurations or are specific to DTLS. McMahon Stone et al. [247] extend state learning to capture time behavior and operate over an unreliable communication medium. They then used their extension to analyze the 802.11 4-Way Handshake implementations in seven Wi-Fi routers. In dealing with non-determinism, our work employs some of the same strategies, such as checking counterexamples against a cache or using majority voting. However, it can use a more efficient learning setup, as it does not have to deal with a lossy medium and resulting timeouts. Chalupar et al. [248] also had to address non-determinism of the system, though this time it was not introduced by the medium but by the system itself. In their work, a simple majority voting system was sufficient to address these issues.

Ecosystem. As established in Chapter 9, the analysis of the TLS ecosystem is of high importance to research. The most important research line in this area is the series of papers around ZMap and ZGrab [157,159,249,250]. The techniques in these papers were used in countless studies on TLS and other protocols. A common target with these scanners is the certificate ecosystem, which were analyzed by Kumar et al. [251], VanderSloot et al. [252] and Durumeric et al [253]. Springall et al. analyzed ephemeral key reuse, STEKs and session cache longevity. A study on the importance of the geographical location of the scanner was conducted by Wan et al [254]. Simoiu et al. [255] analyzed basic information, like supported cipher suites or protocol versions to estimate the security of the ecosystem. Henninger et al. [256] analyzed an interesting vulnerability related to weak public keys that can only be uncovered without implementation details by performing Internet-wide scans. Durumeric et al. [257] analyzed the impact of the Heartbleed vulnerability with Internet-wide scans. Holz et al. [133] and Mayer et al. [258] were the first who considered ports beyond 443; they evaluated the security of TLS for SMTP, IMAP, or XMPP. Valenta et al. [118] measured the impact of small subgroup attacks. Besides the aforementioned studies on the ecosystem, it is also common practice to estimate the impact of vulnerabilities with Internet wide scans, as was done by Adrian et al. [115], Aviram et al [93], Böck et al. [93], and Valenta et al [259]. A different approach to ecosystem analysis was taken by Kotzias et al. [260], where the authors passively analyzed network traffic over several years for negotiated parameters

Post Quantum TLS. The potential threat of upcoming quantum computers has led the community towards new algorithms that are also considered secure if the adversaries have access to a quantum computer. The integration of these algorithms into real-world protocols has been a major concern, as different aspects of the protocol interplay with these algorithms. A lot of work in this area is done by Schwabe, Stebila, Paquin, Crockett and Wiggers [261–264].

Hidden Number Problem. The HNP has been applied in many attacks against DSA, and ECDSA with partially known nonces and signatures in zero-knowledge proofs, using a variety of side channels [43, 44, 47, 265, 266]. Breitner and Heninger [41] used a lattice-based HNP solver to compute private ECDSA signing keys generated by cryptocurrency code.

Some previous attacks on the TLS protocols also have made use of lattice techniques and HNP solvers. Brumley and Tuveri [211] exploited a timing side channel in the ECDSA signature generation during the TLS handshake.

Cross-Protocol Attacks. Early cross-protocol attacks were found in cryptographic systems. Kelsey, Schneier, and Wagner [267] described how new protocols can be designed to allow such attacks on existing protocols, foreshadowing some of the problems occurring when key material, certificates, or cryptographic protocols (such as TLS) are reused for different applications. They also gave basic principles for protocol design to avoid such issues. Their work was expanded by Canetti et al. [136], who considered the environmental requirements for authentication protocols and showed that even strong protocols can fail for external reasons. For TLS, cryptographic cross-protocol attacks were examined by Wagner and Schneier [196], Mavrogiannopoulos et al. [268] and Aviram et al. [93] (DROWN). Nir and Gueron [269] analyzed a similar weakness in TLS 1.3 PSK where the client is running a server with the same pre-shared key as the intended server. In their 'Selfie' attack, the attacker redirects the messages from the client C back to its own server without the client noticing that confusion.

The first example of an application layer cross-protocol attack was described by Topf [270], who showed how to send emails via SMTP over HTTP from an HTML form. He recognized this as a way to access intranet services behind a firewall. The first systematization of these attacks was provided by Alcorn [271] and subsequently extended to demonstrate the impact on internal networks behind firewalls [272] (*inter-protocol exploitation*) by giving an attack on the Asterisk Manager Interface through HTTP. Other authors applied these techniques to attack UPnP [273], FTP [274], IMAP [275], and Redis [276] servers in internal networks, often paired with other vulnerability exploits to achieve remote code execution. Other works discussed how to send commands to network printers [277] or spam [278] from HTML websites. The most recent summary was given by Prynn [279] who also showed how HTTP can be combined with binary protocols. Common to these works is that they give no consideration to TLS and that they are attacking the protocol wrapped inside the HTTP protocol, rather than the HTTP server. These attacks were also considered in the design of HTTP/2 [280].

A simple XSS (Cross Site Scripting) attack against web servers with colocated services vulnerable to reflection attacks was described by Gauci [281, 282]. The first structured presentation of the XSS attack scenario was presented by Alcorn [271]. Horn presented the first example for a JavaScript download attack using FTP over HTTPS as a MitM attacker in a bug report against the *ProFTPD* FTP sever [14], which is the first time that TLS application data confusion was used to enable cross-protocol attacks. Horn also pointed out a vulnerability in vsftpd and a potential vulnerability in Dovecot IMAP and suggested the use of ALPN to mitigate the attacks [283]. In their study on printer attacks, Müller et al. [284] showed how to use cross-site printing to spoof CORS (Cross-Origin Resource Sharing[1]) headers and thus get access to data from a different origin.

Another line of research considered attacks on TLS application data confusion within the same protocol, rather than different protocols. Delignat-Lavaud and Bhargavan [134] analyzed how a MitM can exploit HTTPS virtual hosting configurations, and Zhang et al. [135] found even more HTTPS MitM attacks, exploiting insecure web security policies in the substitute server or mixing TLS with plaintext content.

Fuzzing. American Fuzzy Lop (AFL) [285], AFL++ [286], and honggfuzz [287] are general-purpose fuzzing tools. They execute specific functions or programs with mutated inputs and employ a sophisticated strategy that combines different techniques to discover new code paths.

AFLnet [16] introduces network capabilities to `AFL` and turns it into a feedback-driven network protocol fuzzer that employs code coverage and state information as a feedback mechanism. While this works well for some network protocols, our evaluation demonstrates that, in just a few minutes of fuzzing, our highly-specific TLS protocol fuzzer surpasses the total number of edges covered by AFLnet after 22-24h of fuzzing. Furthermore, our communication synchronization mechanism allows us to avoid unnecessary timeouts.

`Nyx-Net` [17] is a network fuzzing tool based on `Nyx` [184] that introduces snapshots to improve the speed of network fuzzing. Additionally, `Nyx-Net` emulates the network functionality to reduce the overhead and predict timeouts, similar to `network-emulator` [174] and our own communication synchronization solution. Although `Nyx-Net` is not TLS aware, a combination of ETF's in-depth TLS knowledge and `Nyx-Net`'s snapshot-based full-system fuzzing would likely bring additional speedups.

[1] `https://fetch.spec.whatwg.org/`

13 Future Work

The work presented in this dissertation could be extended in a lot of different directions.

Raccoon. While we demonstrated that the Raccoon attack is generally possible, we could not solve the HNP for all interesting cases. Most notably, we were unable to solve the HNP for 2048 to 4096-bit moduli with a leak of $k = 8$. Solving these equation systems would be a great improvement to the attack. Another interesting research question is to which extent other protocols and applications are also affected by the Raccoon attack.

Alpaca. The Alpaca attack only scratched the surface of cross-protocol attacks and can be extended in different directions. The original publication has only studied cross-protocol attacks of HTTPS on FTP and email protocols. Other cross-protocol attack scenarios and protocol combinations need to be analyzed. This does not only include text-based protocols, as similar cross-protocol attacks can also be applied to binary protocols. Our attacks work because of the lack of authentication between TLS and the application layer protocols. Similar problems can arise in other cryptographic protocols, such as DTLS [7] or IPsec [288].

TLS-Attacker. The TLS-Attacker framework has already grown very mature over the past years, however, there are still a lot of possible extensions. Generally speaking, lesser used features offer more potential for bugs since they are generally not tested that often, which is why it may be worth extending TLS-Attacker in that regard. A list of potential features that could be added to TLS-Attacker is presented in Chapter A. Another promising direction for TLS-Attacker is to move more towards TLS deviates, or even completely away from TLS to other protocols like SSH, Kerberos, or IPsec. The tooling support TLS-Attacker provided proved very valuable although TLS was already a very well-researched protocol. Having comparable tooling for other protocols would open up an enormous research space.

Ecosystem. Analysis of the TLS ecosystem is an ongoing process and will always be of academic interest. However, besides the repetition of previous studies, there are a lot of unexplored aspects of the TLS protocol. For example, for recent PQC developments, it is very important to understand the extent of intolerances and other bugs that could hinder the deployment of TLS-PQC algorithms. Also, it is an interesting open question how the security community can better help server operators detect and remediate more subtle vulnerabilities. During our disclosure of various vulnerabilities, we noticed that many security teams of affected companies struggled to understand our reports and often could not forward us to the vendor of their TLS implementation. In the SSH and IPsec protocols, this information is typically transmitted as message fields in the protocol, such that this issue does not emerge. Transmitting such data in TLS would make disclosure easier. An alternative approach could be to perform fingerprinting to 'guess' the vendor of a given TLS implementation. Reliable fingerprinting for TLS has never been achieved on a significant scale.

Evolutionary TLS Fuzzing. One of the major shortcomings of ETF is that the analyzer does not understand the semantics behind corrupted messages ETF generates. If ETF had an objectively correct view of the exchanged message flows, the analyzer could find more complex semantic issues. Although ETF has a lower number of executions per second than AFLnet, it could still reach more code coverage in significantly less time. As such, any future work on improving the speed of ETF will likely be reflected in its fuzzing performance. Other fuzzing techniques like snapshot fuzzing [17, 184] could also help ETF to execute faster, especially since ETF is currently not taking advantage of advances in process restarting mechanisms.

Another interesting research direction could cover other architectures and compiler flags. TLS implementations are primarily configured with compile-time flags that enable or disable entire features or switch the code flow to different implementations (e.g., with different CPU/memory tradeoffs). Security researchers do not as commonly analyze these non-default configurations, making them an excellent target for future research.

State Machine Fuzzing. There are several directions to extend our approach to state learning. The most important extension would be further automation using model checking techniques. This could, for example, be done by performing similar work as was done for TCP or SSH by Broştean et al. [289, 290]. Furthermore, our work only considered valid messages, it would be interesting to evaluate how carefully selected *invalid* messages improve the technique. Yet another extension of the approach could focus on the generation of test cases from the learned model to test properties of the implementation.

14 Conclusions

This dissertation covers a broad range of topics related to TLS that improved the state of the TLS ecosystem from multiple angles.

Raccoon Attack. The Raccoon attack presented a new timing side channel on TLS-DH(E) within the TLS specification, similar to the Lucky13 [11] attack. While the HNP [13] was well known for many years (also in the context of DH), it was scientifically unknown how it could be applied in real-world protocols. The Raccoon attack introduced a new potential side-channel pit-fall that is also of concern for other protocols beyond TLS. To many, the attack was a surprise, as both TLS-DH and TLS-DHE have been proven secure under the PRF-ODH assumption [215,216]. As the Raccoon attack could show, real-world implementations do not meet the PRF-ODH assumption due to the construction of the KDF. The Raccoon attack, in that sense, is a prime motivator for more collaboration to evaluate if real-world implementations meet the assumptions within the theoretical models of the protocol.

Alpaca. With the Alpaca attack, a flaw in the authentication of TLS was exploited that was already known to some experts in the field. However, the flaw was not well understood; neither the component responsible for the issue was identified nor the extent to which the vulnerability was exploitable. The Alpaca attack answered both of these questions and put a spotlight on the issue, such that a discussion could be started on how the issue could be fixed. While some TLS libraries mitigated the issue in newer versions, other libraries are more conservative with their countermeasures and require that the application using the library actively enables the countermeasure. The activation is often non-trivial, which will very likely slow down the deployment of effective countermeasures.

TLS-Attacker. The TLS-Attacker framework proves very helpful, not only for the development of testing frameworks but also for rapid prototyping of attacks and building tools to analyze the ecosystem. Complete feature scale evaluations required days and not months to conduct, which was very valuable during my research. One of the significant advances for the future of TLS-

Attacker will be the extension to protocols beyond TLS like SSH, QUIC, or IPsec, which will benefit from the years of research on TLS and the experience gained throughout the development of TLS-Attacker. One of the shortcomings of TLS-Attacker is high-performance applications. TLS-Attacker is designed for rapid development and flexibility and not performance. In some applications, this becomes a genuine concern. For example, in fuzzing, high performance is crucial to achieving good code coverage. Additionally, some attacks require the fast transmission of a lot of bulk data, which is impossible with the design of TLS-Attacker. This can be a significant disadvantage when performance analysis is done for tools and attacks that rely on TLS-Attacker.

TLS-Scanner. Before TLS-Scanner was released, the open-source TLS scanner landscape was mostly dominated by simple tools that could query supported cipher suites, protocol versions, or the used certificate. One notable exception to this was 'testssl.sh' [141]. However, testssl.sh was limited by its design, purely written in bash. It, therefore, relies on the availability of certain APIs in OpenSSL to perform its tests. In contrast, TLS-Scanner relies on TLS-Attacker as the underlying TLS implementation, which has access to effectively all aspects of the TLS connection. This allows TLS-Scanner to create sophisticated tests that do not rely on 'normal' APIs. For example, the `PaddingOracleProbe` (see Subsection 8.2.2) would be very challenging to implement without tools like TLS-Attacker. The TLS-Scanner project was beneficial for academic research as it allowed for the development of more advanced tools. Commonly, these tools need to know which features an implementation supports to achieve full automation. For this purpose, it is beneficial to have a tool that can automatically extract the required information. This automation cuts down a lot of the configuration time and directly improves any newly developed research technique. Additionally, the flexibility of TLS-Scanner is ideal for evaluating new attacks or quantifying observed phenomena like the Alpaca or Raccoon attack, as the evaluation could be done within mere days.

TLS-Crawler. With TLS-Crawler, it is possible to spread the load of the scanning of TLS servers horizontally. The tool does an excellent job for most research purposes, as it allows scaling up classically programmed `Probes` without additional work from the user. The obvious drawback of this flexibility is its speed, which is slow compared to other tools like ZMap [159] or ZGrab [157]. However, for many research projects we did, it was not necessary to scan the whole IPv4 address space, but enough to scan a sufficiently large subset. The slowdown of the scanning process was quickly compensated by using more computing resources, which dramatically reduced the human resources necessary for the projects.

Ecosystem. During my work on TLS-Attacker and TLS-Scanner, I conducted various studies of the TLS ecosystem, some of which are presented in this dissertation, most of them performed on HTTPS. In my evaluation of CBC padding oracle vulnerabilities in 2019, only 1.83 % of servers were vulnerable to padding

oracles that did not require a timing side channel. Only 57.5% of those servers were exploitable (again without using timing side channels). Overall, the study results indicate that only a small fraction of the servers are vulnerable to this more severe attack. Many more vulnerabilities are likely present that only manifest in timing side channels. While cryptographers often worry a lot about timing side channels, for practitioners, these vulnerabilities are usually very low on the threat scale if they are not exploitable remotely, as they are tough to execute successfully outside of a lab environment. Additionally, these vulnerabilities are only a threat if a CBC cipher suite is used in the connection that will be attacked. This drastically reduces the likelihood of a successful attack and mostly limits the attack to legacy systems.

Similarly, the evaluation of the Raccoon attack revealed that most servers are not reusing DH keys anymore, preventing the attack from being exploitable. This would have been different a few years ago as the most prominent implementation, OpenSSL, was reusing DH parameters by default. Since the Raccoon attack is also primarily a timing vulnerability, the impact on the overall ecosystem and the real security of the affected systems is probably low. For the TLS ecosystem, both results are good news. As practically exploitable vulnerabilities seem to be rare.

The Alpaca vulnerability is different in that regard, as the overall vulnerability is common, and the exploitation is comparably trivial. However, the Alpaca attack requires additional support from the application layer to exfiltrate useful information. Practical exploitation techniques were investigated in the full publication [10] and showed that many servers are at risk.

All in all, the TLS ecosystem is in reasonably good shape. Although many potential threats have to be mitigated, most servers are dealing pretty well with this constantly evolving threat landscape.

TLS-Fuzzing. Fuzzing TLS libraries is challenging. Many current libraries add additional code to the libraries to ease fuzzing for traditional fuzzers. This added code is very dangerous if ever deployed in a production environment as it disables many security features like MAC or signature verifications. Additionally, the library developer must annotate all critical locations; or otherwise, the fuzzer will miss important code paths. ETF presented an alternative to this paradigm, where the fuzzer does not require this additional guidance and can rely on coverage information from AFL. The fuzzer unsurprisingly outperformed simple context unaware fuzzers and was comparable to TLS test suites like tlsfuzzer [179]. A significant drawback of ETF is the dependency on a framework like TLS-Attacker to create its inputs. If TLS-Attacker does not support all the features supported by the SUT, the chances of ETF achieving relevant code coverage in said features fastly diminish.

(D)TLS-State-Learning. Yet another tool built upon TLS-Attacker is the state machine fuzzer presented in Chapter 11. The tool uses a black-box approach to extract the state machine from a (D)TLS implementation as a Mealy machine. The technique brought great results and discovered severe vulnera-

bilities in real-world implementations, the most prominent being the client authentication bypass in JSSE. The general approach is scientifically interesting as it provides a methodology to automatically discover previously unknown *logic* flaws without explicitly defining the exact test cases used to discover the vulnerability. Model checking could be used to define a rule set of unwanted behavior in a state machine, which could then be automatically checked. In the context of TLS, this is a huge step forward from previously applied testing strategies, as they often required explicitly defined test cases or could only find classical programming errors. While the techniques used in the state machine fuzzer are still limited as they are restricted by the alphabet used during learning, they are a huge step towards fully autonomous logical vulnerability discovery.

Closing remarks. The TLS protocol is a unique protocol and one of the big achievements of cryptography. The protocol was one of the first to make cryptography widely available to ordinary people. While the protocol was never flawless, the overall security level is widely perceived as very high. With my dissertation, I contributed to its legacy by uncovering previously unknown flaws, even further hardening the protocol. With the tools and analysis techniques I developed, I reduced the barrier to entry for new practical research and uncovered multiple serious flaws in popular real-world implementations. I hope that future research will further extend the work done on the TLS-Attacker project to other protocols such they can eventually receive the same level of scrutiny and accessibility as TLS.

Acknowledgements

This work would not have been possible without the help of my friends, colleagues, and supervisors. First, I would like to thank my supervisor Prof. Dr. Jörg Schwenk, who put a lot of trust in me and my research. He provided me with the freedom and resources (in time, hardware, and personnel) to pursue our joint vision of the project. I also highly value his experience in the presentation of research results and the fruitful discussions, which certainly more than once prevented our papers from being rejected. Next, I would like to thank Prof. Dr. Juraj Somorovsky who unofficially co-supervised me, especially in the first years of my dissertation. He taught me a lot about TLS, leadership, and academic practices and for that, I will be forever thankful. I also want to thank my colleague and friend Dr. Paul Rösler, with whom I shared an office for many years. Our wild discussions heavily influenced my view on cryptography, academic research, and the world. Our shared late nights in the office pushed me to work harder on my goals, for which I am very grateful. I would like to thank my colleague and friend Marcus Brinkmann, with whom I also had countless discussions. I highly value his experience and opinions. I learned a lot from our cooperation and changed the way I see the world for the better. I would like to thank my colleagues, Marcel Maehren, Nurullah Erinola, Fabian Bäumer, and Sven Hebrok who are super fun and easy to work with and share with me a joint vision of the future of the field. I would like to thank all my co-authors, especially Nimrod Aviram who brought me down to earth more than once when I got too excited about a new discovery.

Additionally, I would like to thank my current and former student assistants who worked with me on TLS-Attacker: Niklas Niere, Markus Hecker, Jia-Lin Shi, Johann Voigt, Luca Bergfort, Pierre Tilhaus, Tim Strom, Selami Hoxha, Henrik Schaefer and Nils Klümper. Without their help, it would be a lot harder to develop and maintain such a big project.

I would also like to thank all the students I had the honor of supervising their Bachelor's and Master's theses. Their work enabled me to tackle my research challenges much broader and faster than I could have done on my own.

Last but not least I would like to thank my family, especially my loving wife Vanessa, who unconditionally supported me throughout my career which allowed me to focus on my work. I would also like to thank my son Leander, your contagious smile brightened my days while I wrote this dissertation.
I love you.

A

TLS-Attacker Features

Feature	TLS-Attacker 1.2	TLS-Attacker 4.8
Client Support	●	●
Server Support	◐	●
Client Authentication	●	●
Renegotiation (Client Side)	○	●
Renegotiation (Server Side)	○	●
Session Resumption (ID)	○	●
Session Resumption (Ticket)	○	●
0-RTT	○	●
Key-Update	○	●
Man-in-the-Middle	◐	●
EAP	◐	○
Post-Handshake Authentication (TLS 1.3)	○	○

○: Not supported feature ●: Supported feature ◐: Partially supported feature.

Table A.1.: An overview of general TLS-Attacker features.

Version	TLS-Attacker 1.2	TLS-Attacker 4.8
SSLv2	○	◐
SSLv3	○	●
TLS 1.0	●	●
TLS 1.1	●	●
TLS 1.2	●	●
TLS 1.3	○	●
DTLS 1.0	○	●
DTLS 1.2	◐	●
DTLS 1.3	○	○

○: Not supported feature ●: Supported feature ◐: Partially supported feature.

Table A.2.: An overview of protocol versions supported by TLS-Attacker.

Algorithm	TLS-Attacker 1.2	TLS-Attacker 4.8
NULL	○	●
RC2-CBC	○	●
RC2-CBC-Export	○	○
RC4	○	●
RC4-Export	○	○
DES-CBC	○	●
DES-CBC-Export	○	○
3DES-CBC	●	●
AES-128-CBC	●	●
AES-256-CBC	●	●
AES-128-GCM	○	●
AES-256-GCM	○	●
AES-128-CCM	○	●
AES-256-CCM	○	●
CAMELLIA-128-CBC	○	●
CAMELLIA-256-CBC	○	●
CAMELLIA-128-GCM	○	●
CAMELLIA-256-GCM	○	●
IDEA-CBC	○	●
SEED-CBC	○	●
Chacha-Poly1305 (RFC 7905)	○	●
Chacha-Poly1305 (Unofficial)	○	●
ARIA-128-CBC	○	●
ARIA-256-CBC	○	●
ARIA-128-GCM	○	●
ARIA-256-GCM	○	●
GOST-28147-CNT	○	◐
FORTEZZA	○	○

○: Not supported feature ●: Supported feature ◐: Partially supported feature.

Table A.3.: An overview of supported bulk encryption algorithms in TLS-Attacker.

Algorithm	TLS-Attacker 1.2	TLS-Attacker 4.8
ECDHE-ECDSA	●	●
ECDHE-RSA	●	●
ECDH-RSA	●	●
ECDH-ANON	○	●
ECDH-ECDSA	●	●
DHE-DSS	●	●
DHE-DSS-Export	●	●
DHE-RSA	●	●
DH-anon	○	●
DH-DSS	●	●
DH-RSA	●	●
RSA	●	●
Export (RSA)	○	●
Export (DH)	◐	●
PSK	○	●
PSK-RSA	○	●
DHE-PSK	○	●
ECDHE-PSK	○	●
KRB5	○	○
SRP_SHA_DSS	○	●
SRP_SHA_RSA	○	●
SRP_SHA	○	●
VKO-GOST01	○	◐
VKO-GOST12	○	◐
FORTEZZA-KEA	○	○
ECMQV-ECDSA	○	○
ECMQV-ECNRA	○	○
ECDH-ECNRA	○	○
CECPQ1_ECDSA	○	○
ECCPWD	○	●

○: Not supported feature ●: Supported feature ◐: Partially supported feature.

Table A.4.: An overview of supported key exchange algorithms in TLS-Attacker.

Format	**TLS-Attacker 1.2**	**TLS-Attacker 4.8**
uncompressed	●	●
ansiX962_compressed_prime	○	●
ansiX962_compressed_char2	○	●

○: Not supported feature ●: Supported feature ◐: Partially supported feature.

Table A.5.: An overview of supported EC point formats in TLS-Attacker.

Algorithm	TLS-Attacker 1.2	TLS-Attacker 4.8
ANONYMOUS	○	●
RSA-PKCS1-MD5	●	●
RSA-PKCS1-SHA	●	●
RSA-PKCS1-SHA224	●	●
RSA-PKCS1-SHA256	●	●
RSA-PKCS1-SHA384	●	●
RSA-PKCS1-SHA512	●	●
ECDSA-SHA1	●	●
ECDSA-SHA224	●	●
ECDSA-SHA256	●	●
ECDSA-SHA384	●	●
ECDSA-SHA512	●	●
RSA-PSS-SHA256	○	●
RSA-PSS-SHA384	○	●
RSA-PSS-SHA512	○	●
ED25519	○	●
ED448	○	●
DSA-SHA1	●	●
DSA-SHA224	●	●
DSA-SHA256	●	●
DSA-SHA384	●	●
DSA-SHA512	●	●
GOSTR34102001-GOSTR3411	○	◐
GOSTR34102012-256-34112012_256	○	◐
GOSTR34102012-512-34112012_512	○	◐
ECCSI SHA256	○	○
ISO IBS1	○	○
ISO IBS2	○	○
ISO Chinese IBS	○	○
SM2 Sig SM3	○	○
GOSTR34102012-256A	○	○
GOSTR34102012-256B	○	○
GOSTR34102012-256C	○	○
GOSTR34102012-256D	○	○
GOSTR34102012-512A	○	○
GOSTR34102012-512B	○	○
GOSTR34102012-512C	○	○
RSA PSS-PSS SHA256	○	○
RSA PSS-PSS SHA384	○	○
RSA PSS-PSS SHA512	○	○
ECDSA Brainpool P256R1SHA256 TLS 1.3	○	○
ECDSA Brainpool P384R1SHA384 TLS 1.3	○	○
ECDSA Brainpool P512R1SHA512 TLS 1.3	○	○

○: Not supported feature ●: Supported feature ◐: Partially supported feature.

Table A.6.: An overview of supported signature and hash algorithms in TLS-Attacker.

Extension	TLS-Attacker 1.2	TLS-Attacker 4.8
Server Named Indication (SNI)	●	●
Max Fragment Length	●	●
Client Certificate URL	○	●
Trusted CA Keys	○	●
Truncated HMAC	○	◐
Status Request	○	●
User Mapping	○	◐
Client Authz	○	●
Server Authz	○	●
Certificate Type	○	●
Supported Groups	●	●
EC Point Formats	●	●
SRP	○	●
Signature Algorithms	●	●
Use SRTP	○	●
Heartbeat	●	●
ALPN	○	●
Status Request v2	○	●
Signed Certificate Timestamp	○	●
Client Certificate Type	○	●
Server Certificate Type	○	●
Padding	○	●
Encrypt-then-MAC	○	●
Extended Master Secret	○	●
Token Binding	○	●
Cached Info	○	◐
TLS LTS	○	○
Certificate Compression	○	○
Record Size Limit	○	●
PWD Protect	○	●
PWD Clear	○	●
Password Salt	○	●
Ticket Pinning	○	○
Cert with Extern PSK	○	○
Delegate Credentials	○	○
Session Ticket	○	●
TLMSP	○	○
TLMSP Proxying	○	○
TLMSP Delegate	○	○
Supported EKT Ciphers	○	○
Preshared Key	○	●
Early Data	○	●
Supported Versions	○	●
Cookie	○	●
PSK Key Exchange Modes	○	●
Certificate Authorities	○	○
OID Filters	○	○
Post Handshake Authentication	○	○
Signature Algorithms (Certificate)	○	○
Key Share	○	●
Transparency Info	○	○
Connection ID	○	○
External ID Hash	○	○
External Session ID	○	○
QUIC Transport Parameters	○	○
Ticket Request	○	○
DNSSEC Chain	○	○
Renegotiation Info	○	●

○: Not supported feature ●: Supported feature ◐: Partially supported feature.

Table A.7.: An overview of supported extensions in TLS-Attacker.

B

Fuzzer

Library	Version	Configuration
BoringSSL	Commit 49e9f67d	bssl s_server -accept port -key tlskey -cert tlscert -loop -www
Botan	2.15.0	botan tls_server tlscert tlskey –port=port –policy=botan/src/tests/data/tls-policy/compat.txt –max-clients=0
GnuTLS	3.7.0	gnutls-serv –x509keyfile tlskey –x509certfile tlscert -p port -a –http
MatrixSSL	4-3-0-open	server
mbedTLS	2.24.0	ssl_server2 crt_file=tlscert key_file=tlskey server_port=port
OpenSSL	3.0.0	openssl s_server -port port -key tlskey -cert tlscert -www
wolfSSL	v4.5.0-stable	server -p port -d -k tlskey -c tlscert -x -i -r

Table B.1.: Configuration parameters of all libraries for the code coverage evaluation.

Library	Version	Configuration
BoringSSL	Commit 433c0aab	bssl client -connect localhost:port
Botan	Commit f3506ee5	botan tls_client localhost –port=port
GnuTLS	Commit 67f7df09	gnutls-cli localhost -p port
MatrixSSL	3-9-3-open	client -p port
mbedTLS	2.6.0	ssl_client2 server_port=port
OpenSSL	1.1.0f	openssl s_client -connect localhost:port
wolfSSL	Commit e43e03c3	client -p port -d

Table B.2.: Configuration parameters of all libraries for the bug finding evaluation / case study.

Tool	Commit	Changes	Configuration
ETF	84040d73	bitmap size 1048576	java -jar ./target/EvolutionaryFuzzer-1.00.jar serverfuzzer -agent AFL -analyzer RULE -mutator ADAPTIVE -quickReceive -skipCalibration -skipSelftest -output outputdir/ -afl_bitmapsize 1048576
AFLNET	8e0800da	bitmap size 1048576	AFL_NO_AFFINITY=1 afl-fuzz -i inputdir -o outputdir -d -N tcp://127.0.0.1/port -P TLS -q 3 -s 3 -E -K -m none -t 40000 -W 30 -R –binarycommand
TLS-ATTACKER	951c3d3e		java -jar TLS-Attacker-1.2.jar multi_fuzzer -startup_command_file xml
TLSFUZZER	cde50168		<script>.py -h 127.0.0.1 -p fuzzerport
TLSBUNNY	6577817b	disable sefltest default hash	java -cp target/tlsbunny-1.0-SNAPSHOT-all.jar com.gypsyengineer.tlsbunny.tls13.client.fuzzer.DeepHandshakeFuzzyClient standard
AFL-SHOWMAP	8e0800da	Data collection bitmap size 1048576	afl-showmap -o outputdir/out -m none -t 4000

Table B.3.: Configuration parameters of all fuzzers.

List of Figures

List of Tables

Bibliography

[1] P. Hoffman, "SMTP Service Extension for Secure SMTP over Transport Layer Security," RFC 3207 (Proposed Standard), RFC Editor, Fremont, CA, USA, Feb. 2002, updated by RFC 7817. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3207.txt

[2] C. Newman, "Using TLS with IMAP, POP3 and ACAP," RFC 2595 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jun. 1999, updated by RFCs 4616, 7817, 8314. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2595.txt

[3] Y. Rekhter (Ed.), T. Li (Ed.), and S. Hares (Ed.), "A Border Gateway Protocol 4 (BGP-4)," RFC 4271 (Draft Standard), RFC Editor, Fremont, CA, USA, Jan. 2006, updated by RFCs 6286, 6608, 6793, 7606, 7607, 7705, 8212, 8654, 9072. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4271.txt

[4] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)," RFC 7858 (Proposed Standard), RFC Editor, Fremont, CA, USA, May 2016, updated by RFC 8310. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7858.txt

[5] P. Hoffman and P. McManus, "DNS Queries over HTTPS (DoH)," RFC 8484 (Proposed Standard), RFC Editor, Fremont, CA, USA, Oct. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8484.txt

[6] J. Iyengar (Ed.) and M. Thomson (Ed.), "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000 (Proposed Standard), RFC Editor, Fremont, CA, USA, May 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9000.txt

[7] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347 (Proposed Standard), RFC Editor, Fremont, CA,

USA, Jan. 2012, obsoleted by RFC 9147, updated by RFCs 7507, 7905, 8996, 9146. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6347.txt

[8] "Openvpn," 2001. [Online]. Available: https://openvpn.net/

[9] R. Merget, M. Brinkmann, N. Aviram, J. Somorovsky, J. Mittmann, and J. Schwenk, "Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E)," in *Proceedings of the 30th USENIX Security Symposium*, ser. Proceedings of the 30th USENIX Security Symposium. USENIX Association, 2021, pp. 213–230.

[10] M. Brinkmann, C. Dresen, R. Merget, D. Poddebniak, J. Müller, J. Somorovsky, J. Schwenk, and S. Schinzel, "ALPACA: application layer protocol confusion - analyzing and mitigating cracks in TLS authentication," in *USENIX Security Symposium*. USENIX Association, 2021, pp. 4293–4310.

[11] N. J. AlFardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in *2013 IEEE symposium on security and privacy*. IEEE, 2013, pp. 526–540.

[12] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1," in *Advances in Cryptology — CRYPTO '98*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1998, vol. 1462.

[13] D. Boneh and R. Venkatesan, "Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes," in *Advances in Cryptology — CRYPTO '96*. Springer Berlin Heidelberg, 1996, pp. 129–142. [Online]. Available: https://doi.org/10.1007%2F3-540-68697-5_11

[14] J. Horn, "HTTPS/FTPS protocol confusion leads to XSS (ProFTP Bug 4143)," 2014. [Online]. Available: http://bugs.proftpd.org/show_bug.cgi?id=4143#c0

[15] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1492–1504. [Online]. Available: https://doi.org/10.1145/2976749.2978411

[16] V.-T. Pham, M. Bohme, and A. Roychoudhury, "AFLNET: a greybox fuzzer for network protocols," in *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation, ICST 2020*, C. Pasareanu and A. Zeller, Eds. United States of America: IEEE, Institute of Electrical and Electronics Engineers, 2020. [Online]. Available: https://ieeexplore.ieee.org/xpl/conhome/9149738/proceeding, https://icst2020.info

[17] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: Network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22, 2022.

[18] G. Vranken, "Cryptofuzz," 2019. [Online]. Available: https://github.com/guidovranken/cryptofuzz

[19] R. Merget, J. Somorovsky, N. Aviram, C. Young, J. Fliegenschmidt, J. Schwenk, and Y. Shavitt, "Scalable scanning and automatic classification of TLS padding oracle vulnerabilities," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1029–1046. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/merget

[20] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2523–2540. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[21] S. Lauer, K. Gellert, R. Merget, T. Handirk, and J. Schwenk, "T0RTT: non-interactive immediate forward-secret single-pass circuit construction," *Proc. Priv. Enhancing Technol.*, vol. 2020, no. 2, pp. 336–357, 2020.

[22] M. Maehren, P. Nieting, S. Hebrok, R. Merget, J. Somorovsky, and J. Schwenk, "TLS-Anvil: Adapting combinatorial testing for TLS libraries," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 215–232. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/maehren

[23] R. Fielding (Ed.) and J. Reschke (Ed.), "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing," RFC 7230 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jun. 2014, obsoleted by RFCs 9110, 9112, updated by RFC 8615. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7230.txt

[24] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '98. Berlin, Heidelberg: Springer-Verlag, 1998, p. 26–45.

[25] T. Duong and J. Rizzo, "Here come the ⊕ Ninjas," in *EKOparty*, 2011.

[26] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 199–212.

[27] M. Bellare and P. Rogaway, "Introduction to modern cryptography," in *UCSD CSE 207 Course Notes*, 2005.

[28] D. J. Bernstein *et al.*, "Chacha, a variant of salsa20," in *Workshop record of SASC*, vol. 8, no. 1. Lausanne, Switzerland, 2008, pp. 3–5.

[29] M. Dworkin, "Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac," 2007-11-28 2007. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=51288

[30] National Institute of Standards and Technology, "FIPS PUB 113: Standard for computer data authentication," National Institute of Standards and Technology, Tech. Rep., May 1985. [Online]. Available: http://www.itl.nist.gov/fipspubs/fip113.htm

[31] I. B. Damgård, "A design principle for hash functions," in *Advances in Cryptology — CRYPTO' 89 Proceedings*, G. Brassard, Ed. New York, NY: Springer New York, 1990, pp. 416–427.

[32] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *Advances in Cryptology – CRYPTO 2010*, T. Rabin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 631–648.

[33] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, p. 120–126, feb 1978. [Online]. Available: https://doi.org/10.1145/359340.359342

[34] D. Bleichenbacher and A. May, "New attacks on RSA with small secret CRT-exponents," in *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, ser. Lecture Notes in Computer Science, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958. Springer, 2006, pp. 1–13. [Online]. Available: https://doi.org/10.1007/11745853_1

[35] A. Delignat-Lavaud, "CVE-2014-1569." Available from MITRE, CVE-ID CVE-2014-1569., May 2016. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1569

[36] B. Kaliski, "PKCS #1: RSA Encryption Version 1.5," RFC 2313 (Informational), RFC Editor, Fremont, CA, USA, Mar. 1998, obsoleted by RFC 2437. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2313.txt

[37] B. Kaliski and J. Staddon, "PKCS #1: RSA Cryptography Specifications Version 2.0," RFC 2437 (Informational), RFC Editor, Fremont, CA, USA, Oct. 1998, obsoleted by RFC 3447. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2437.txt

[38] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, November 1976.

[39] C. F. Kerry, A. Secretary, and C. R. Director, "FIPS PUB 186-4 federal information processing standards publication digital signature standard (DSS)," 2013.

[40] 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.

[41] J. Breitner and N. Heninger, "Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies," in *Financial Cryptography and Data Security*. Springer International Publishing, 2019, pp. 3–20. [Online]. Available: https://doi.org/10.1007%2F978-3-030-32101-7_1

[42] D. F. Aranha, P.-A. Fouque, B. Gérard, J.-G. Kammerer, M. Tibouchi, and J.-C. Zapalowicz, "GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 262–281. [Online]. Available: https://doi.org/10.1007%2F978-3-662-45611-8_14

[43] N. Benger, J. v. de Pol, N. P. Smart, and Y. Yarom, ""ooh aah... just a little bit" : A small amount of side channel can go a long way," in *Advanced Information Systems Engineering*. Springer Berlin Heidelberg, 2014, pp. 75–92. [Online]. Available: https://doi.org/10.1007%2F978-3-662-44709-3_5

[44] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. Volume 2018, pp. Issue 2–, 2018. [Online]. Available: https://ojs.ub.rub.de/index.php/TCHES/article/view/879

[45] E. D. Mulder, M. Hutter, M. E. Marson, and P. Pearson, "Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA: extended version," *Journal of Cryptographic Engineering*, vol. 4, no. 1, pp. 33–45, feb 2014. [Online]. Available: https://doi.org/10.1007%2Fs13389-014-0072-z

[46] P. Q. Nguyen, "The dark side of the hidden number problem: Lattice attacks on DSA," in *Cryptography and Computational Number Theory*. Birkhäuser Basel, 2001, pp. 321–330. [Online]. Available: https://doi.org/10.1007%2F978-3-0348-8295-8_23

[47] P. Q. Nguyen and I. E. Shparlinski, "The insecurity of the digital signature algorithm with partially known nonces," *Journal of Cryptology*, vol. 15, no. 3, pp. 151–176, jun 2002. [Online]. Available: https://doi.org/10.1007%2Fs00145-002-0021-3

[48] A. Takahashi, M. Tibouchi, and M. Abe, "New Bleichenbacher records: Fault attacks on qDSA signatures," *IACR Transactions on Cryptographic*

Hardware and Embedded Systems, vol. Volume 2018, pp. Issue 3–, 2018. [Online]. Available: https://ojs.ub.rub.de/index.php/TCHES/article/view/7278

[49] N. Howgrave-Graham and N. P. Smart, "Lattice attacks on digital signature schemes," *Des. Codes Cryptogr.*, vol. 23, no. 3, pp. 283–290, 2001.

[50] E. D. Mulder, M. Hutter, M. E. Marson, and P. Pearson, "Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2013, pp. 435–452.

[51] A. Akavia, "Solving hidden number problem with one bit oracle and advice," in *Advances in Cryptology - CRYPTO 2009*, S. Halevi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 337–354.

[52] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), RFC Editor, Fremont, CA, USA, Aug. 2008, obsoleted by RFC 8446, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919, 8447, 9155. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5246.txt

[53] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446 (Proposed Standard), RFC Editor, Fremont, CA, USA, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446.txt

[54] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246 (Historic), RFC Editor, Fremont, CA, USA, Jan. 1999, obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465, 7507, 7919. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2246.txt

[55] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346 (Historic), RFC Editor, Fremont, CA, USA, Apr. 2006, obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176, 7465, 7507, 7919. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4346.txt

[56] S. Smyshlyaev (Ed.), D. Belyavsky, and E. Alekseev, "GOST Cipher Suites for Transport Layer Security (TLS) Protocol Version 1.2," RFC 9189 (Informational), RFC Editor, Fremont, CA, USA, Mar. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9189.txt

[57] H. Wu, "A new stream cipher HC-256," in *Fast Software Encryption*, B. Roy and W. Meier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 226–244.

[58] M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen, and O. Scavenius, "Rabbit: A new high-performance stream cipher," in *Fast Software Encryption*, T. Johansson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 307–329.

[59] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280 (Proposed Standard), RFC Editor, Fremont, CA, USA, May 2008, updated by RFCs 6818, 8398, 8399. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5280.txt

[60] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas, "Internet X.509 Public Key Infrastructure: Certification Path Building," RFC 4158 (Informational), RFC Editor, Fremont, CA, USA, Sep. 2005. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4158.txt

[61] P. Gutmann, "Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," RFC 7366 (Proposed Standard), RFC Editor, Fremont, CA, USA, Sep. 2014. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7366.txt

[62] S. Vaudenay, "Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ..." in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, ser. EUROCRYPT '02. Berlin, Heidelberg: Springer-Verlag, 2002, p. 534–546.

[63] B. Möller, T. Duong, and K. Kotowicz, "This POODLE bites: exploiting the SSL 3.0 fallback," *Security Advisory*, vol. 21, pp. 34–58, 2014.

[64] M. Ray and S. Dispensa, "Renegotiating tls," November 2009. [Online]. Available: http://extendedsubset.com/Renegotiating_TLS.pdf

[65] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 98–113.

[66] S. Friedl, A. Popov, A. Langley, and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension," RFC 7301 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jul. 2014, updated by RFC 8447. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7301.txt

[67] D. Eastlake 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions," RFC 6066 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jan. 2011, updated by RFCs 8446, 8449. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6066.txt

[68] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State," RFC 5077 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jan. 2008, obsoleted by RFC 8446, updated by RFC 8447. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5077.txt

[69] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 handshake protocol candidates," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1197–1210. [Online]. Available: https://doi.org/10.1145/2810103.2813653

[70] P. Hoffman, "SMTP Service Extension for Secure SMTP over TLS," RFC 2487 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jan. 1999, obsoleted by RFC 3207. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2487.txt

[71] A. Langley, N. Modadugu, and B. Moeller, "Transport Layer Security (TLS) False Start," RFC 7918 (Informational), RFC Editor, Fremont, CA, USA, Aug. 2016. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7918.txt

[72] G. V. Bard, "The vulnerability of SSL to chosen plaintext attack," *IACR Cryptol. ePrint Arch.*, vol. 2004, p. 111, 2004.

[73] ——, "A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL," in *SECRYPT*, 2006.

[74] B. Möller, "In response to 'an attack against SSH2 protocol'," Email to the ietf-ssh@netbsd.org email list, 2002.

[75] A. Langley, "Beast followup," January 2012. [Online]. Available: https://www.imperialviolet.org/2012/01/15/beastfollowup.html

[76] J. Rizzo and T. Duong, "The CRIME attack," in *Ekoparty Security Conference 2012*, vol. 2012, 2012.

[77] A. S. Tal Be'ery, "A perfect Crime? only time will tell," Blackhat Europe, March 2013.

[78] A. P. Yoel Gluck, Neal Harris, "Breach: Reviving the crime attack," July 2013.

[79] T. V. G. Mathy Vanhoef, "HTTP encrypted information can be stolen through tcp-windows," August 2016.

[80] A. Langley, "The POODLE bites again," Nov. 2014, https://www.imperialviolet.org/2014/12/08/poodleagain.html.

[81] H. Böck, "A little POODLE left in GnuTLS (old versions)," Nov. 2015, https://blog.hboeck.de/archives/877-A-little-POODLE-left-in-GnuTLS-old-versions.html.

[82] Y. Petterssen, "There are more POODLEs in the forest." [Online]. Available: https://yngve.vivaldi.net/2015/07/14/there-are-more-poodles-in-the-forest/

[83] ——, "The POODLE has friends." [Online]. Available: https://yngve.vivaldi.net/2015/07/14/the-poodle-has-friends/

[84] A. W. S. Labs, "s2n: An implementation of the TLS/SSL protocols." [Online]. Available: https://github.com/awslabs/s2n,2015

[85] M. R. Albrecht and K. G. Paterson, "Lucky microseconds: A timing attack on amazon's s2n implementation of TLS," in *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, 2016, pp. 622–643.

[86] E. Ronen, K. G. Paterson, and A. Shamir, "Pseudo constant time implementations of TLS are only pseudo secure," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1397–1414.

[87] J. Somorovsky, "CVE-2016-2107." Available from MITRE, CVE-ID CVE-2016-2107., May 2016. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2107

[88] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic, "Nonce-Disrespecting adversaries: Practical forgery attacks on GCM in TLS," in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/woot16/workshop-program/presentation/bock

[89] H. Böck, J. Somorovsky, and C. Young, "Return of Bleichenbacher's oracle threat (ROBOT)," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 817–849. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bock

[90] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews, "Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks," in *23rd USENIX Security Symposium, San Diego, USA*, August 2014.

[91] T. Jager, S. Schinzel, and J. Somorovsky, "Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption," in *ESORICS*, 2012, pp. 752–769.

[92] R. Bardou, R. Focardi, Y. Kawamoto, G. Steel, and J.-K. Tsay, "Efficient Padding Oracle Attacks on Cryptographic Hardware," in *Advances in Cryptology – CRYPTO*, Canetti and R. Safavi-Naini, Eds., 2012.

[93] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt, "DROWN: Breaking TLS Using SSLv2," in *25th USENIX Security Symposium*

(USENIX Security 16), Austin, TX, Aug. 2016, pp. 689–706. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram

[94] E. Barker and A. Roginsky, "Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths," 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf

[95] P. Rogaway, "Evaluation of some blockcipher modes of operation," 2011.

[96] K. Bhargavan and G. Leurent, "On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 456–467. [Online]. Available: https://doi.org/10.1145/2976749.2978423

[97] ——, "Transcript collision attacks: Breaking authentication in TLS, IKE and SSH," in *NDSS*, 2016.

[98] J. Altman, N. Williams, and L. Zhu, "Channel Bindings for TLS," RFC 5929 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jul. 2010. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5929.txt

[99] I. Biehl, B. Meyer, and V. Müller, "Differential fault attacks on elliptic curve cryptosystems," in *Advances in Cryptology — CRYPTO 2000*, M. Bellare, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 131–146.

[100] T. Jager, J. Schwenk, and J. Somorovsky, "Practical invalid curve attacks on TLS-ECDH." in *ESORICS (1)*, ser. Lecture Notes in Computer Science, G. Pernul, P. Y. A. Ryan, and E. R. Weippl, Eds., vol. 9326. Springer, 2015, pp. 407–425. [Online]. Available: http://dblp.uni-trier.de/db/conf/esorics/esorics2015-1.html#JagerSS15

[101] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 535–552.

[102] D. Boneh, "The decision Diffie-Hellman problem," in *IN THIRD ALGORITHMIC NUMBER THEORY SYMPOSIUM, LNCS 1423*. Springer-Verlag, 1998, pp. 48–63.

[103] J. Brendel, M. Fischlin, F. Günther, and C. Janson, "PRF-ODH: Relations, instantiations, and impossibility results," Cryptology ePrint Archive, Paper 2017/517, 2017, https://eprint.iacr.org/2017/517. [Online]. Available: https://eprint.iacr.org/2017/517

[104] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104 (Informational), RFC Editor, Fremont, CA, USA, Feb. 1997, updated by RFC 6151. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2104.txt

[105] P. Eronen (Ed.) and H. Tschofenig (Ed.), "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)," RFC 4279 (Proposed Standard), RFC Editor, Fremont, CA, USA, Dec. 2005, updated by RFC 8996. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4279.txt

[106] D. Springall, Z. Durumeric, and J. A. Halderman, "Measuring the security harm of TLS crypto shortcuts," in *Proceedings of the 2016 Internet Measurement Conference*, ser. IMC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 33–47. [Online]. Available: https://doi.org/10.1145/2987443.2987480

[107] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 03 1947. [Online]. Available: https://doi.org/10.1214/aoms/1177730491

[108] S. A. Crosby, D. S. Wallach, and R. H. Riedi, "Opportunities and limits of remote timing attacks," *ACM Transactions on Information and System Security*, vol. 12, no. 3, 2009.

[109] A. C. Aldaya and B. B. Brumley, "HyperDegrade: From GHz to MHz effective CPU frequencies," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2801–2818. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/aldaya

[110] Y. Chen and P. Q. Nguyen, "BKZ 2.0: Better lattice security estimates," in *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, ser. Lecture Notes in Computer Science, D. H. Lee and X. Wang, Eds., vol. 7073. Springer, 2011, pp. 1–20. [Online]. Available: https://doi.org/10.1007/978-3-642-25385-0_1

[111] C. P. Schnorr, "A hierarchy of polynomial time lattice basis reduction algorithms," *Theoretical Computer Science*, vol. 53, no. 2-3, pp. 201–224, 1987. [Online]. Available: https://doi.org/10.1016%2F0304-3975%2887%2990064-8

[112] T. F. development team, "fplll, a lattice reduction library," 2016, available at https://github.com/fplll/fplll. [Online]. Available: https://github.com/fplll/fplll

[113] The Sage Developers, *SageMath, the Sage Mathematics Software System (Version 9.1)*, 2020, ttps://www.sagemath.org.

[114] D. Jao, D. Jetchev, and R. Venkatesan, "On the bits of elliptic curve diffie-hellman keys," in *International Conference on Cryptology in India*. Springer, 2007, pp. 33–47.

[115] 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 *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 5–17. [Online]. Available: https://doi.org/10.1145/2810103.2813707

[116] E. T. C. C. Security, "ETSI TS 103 523-3 V1.2.1," https://www.etsi.org/deliver/etsi_ts/103500_103599/10352303/01.02.01_60/ts_10352303v010201p.pdf.

[117] K. G. Paterson and N. J. AlFardan, "Plaintext-recovery attacks against datagram TLS," in *NDSS*, 2012.

[118] L. Valenta, D. Adrian, A. Sanso, S. N. Cohney, J. Fried, M. Hastings, J. A. Halderman, and N. Heninger, "Measuring small subgroup attacks against Diffie-Hellman," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 995, 2017.

[119] A. Adamantiadis, S. Josefsson, and M. Baushke, "Secure Shell (SSH) Key Exchange Method Using Curve25519 and Curve448," RFC 8731 (Proposed Standard), RFC Editor, Fremont, CA, USA, Feb. 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8731.txt

[120] S. G. D. Stebila, S. Fluhrer, "Hybrid key exchange in TLS 1.3." [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-hybrid-design-00

[121] D. Eastlake, J. Reagle, F. Hirsch, T. Roessler, T. Imamura, B. Dillaway, E. Simon, K. Yiu, and M. Nyström, "XML Encryption Syntax and Processing 1.1," *W3C Candidate Recommendation*, 2012, http://www.w3.org/TR/2012/WD-xmlenc-core1-20121018.

[122] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen, "Internet Key Exchange Protocol Version 2 (IKEv2)," RFC 5996 (Proposed Standard), RFC Editor, Fremont, CA, USA, Sep. 2010, obsoleted by RFC 7296, updated by RFCs 5998, 6989. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5996.txt

[123] M. Jones and J. Hildebrand, "JSON Web Encryption (JWE)," RFC 7516 (Proposed Standard), RFC Editor, Fremont, CA, USA, May 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7516.txt

[124] NIST, "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography," Special Publication 800-56A Rev. 3, Apr. 2018.

[125] D. Benjamin, "TLS ecosystem woes," 2018. [Online]. Available: https://drive.google.com/open?id=1jqyTwZlTPD_xp4rTD4FmbsdKYWRHcUkN5lfMeGQZQ_o

[126] A. Freier, P. Karlton, and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0," RFC 6101 (Historic), RFC Editor, Fremont, CA, USA, Aug. 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6101.txt

[127] RSA Laboratories, *PKCS #3: Diffie–Hellman Key-Agreement Standard*, Nov. 1993. [Online]. Available: http://www.rsasecurity.com/rsalabs/pkcs/pkcs-3/index.html

[128] E. Rescorla, "Diffie-Hellman Key Agreement Method," RFC 2631 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jun. 1999. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2631.txt

[129] N. Bolyard, "NSS CKM_DH_PKCS_DERIVE not PKCS3 compliant," https://bugzilla.mozilla.org/show_bug.cgi?id=291498.

[130] K. A. Abluton, "Transport Layer Security (TLS) Handshake Failing, SChannel Error 36888," 2016. [Online]. Available: https://docs.microsoft.com/en-us/archive/blogs/keithab/transport-layer-security-tls-handshake-failing-schannel-error-36888

[131] F5, "K10433354: TLS handshakes from some clients may intermittently fail when using Diffie-Hellman ciphers," 2017. [Online]. Available: https://support.f5.com/csp/article/K10433354

[132] Oracle, "JDK-8014618: Need to strip leading zeros in TlsPremasterSecret of DHKeyAgreement," 2013. [Online]. Available: https://bugs.java.com/bugdatabase/view_bug.do?bug_id=8014618

[133] R. Holz, J. Amann, O. Mehani, M. Wachs, and M. Kâafar, "TLS in the wild: an internet-wide analysis of TLS-based protocols for electronic communication," in *NDSS 2016*, 2016, pp. 1–15, network and Distributed System Security Symposium 2016, NDSS'16 ; Conference date: 21-02-2016 Through 24-02-2016. [Online]. Available: http://www.ndss-symposium.org/ndss2016/

[134] A. Delignat-Lavaud and K. Bhargavan, "Network-based Origin Confusion Attacks against HTTPS Virtual Hosting," in *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, may 2015. [Online]. Available: https://doi.org/10.1145%2F2736277.2741089

[135] M. Zhang, X. Zheng, K. Shen, Z. Kong, C. Lu, Y. Wang, H. Duan, S. Hao, B. Liu, and M. Yang, "Talking with familiar strangers: An empirical study on HTTPS context confusion attacks," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer*

and Communications Security. ACM, oct 2020. [Online]. Available: https://doi.org/10.1145%2F3372297.3417252

[136] R. Canetti, C. Meadows, and P. Syverson, "Environmental Requirements for Authentication Protocols," in *Software Security — Theories and Systems*. Springer Berlin Heidelberg, 2003, pp. 339–355. [Online]. Available: https://doi.org/10.1007%2F3-540-36532-x_21

[137] B. Beurdouche, A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and K. Bhargavan, "FLEXTLS: A tool for testing TLS implementations," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/beurdouche

[138] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-y. Strub, "Implementing TLS with Verified Cryptographic Security," in *IEEE Symposium on Security & Privacy*, San Francisco, United States, 2013, pp. 445–462. [Online]. Available: https://hal.inria.fr/hal-00863373

[139] A. Popov (Ed.), M. Nystroem, D. Balfanz, and J. Hodges, "The Token Binding Protocol Version 1.0," RFC 8471 (Proposed Standard), RFC Editor, Fremont, CA, USA, Oct. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8471.txt

[140] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)," RFC 4492 (Informational), RFC Editor, Fremont, CA, USA, May 2006, obsoleted by RFC 8422, updated by RFCs 5246, 7027, 7919. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4492.txt

[141] D. Wetter, "testssl.sh." [Online]. Available: https://github.com/drwetter/testssl.sh

[142] Qualys, "Ssl labs server test." [Online]. Available: https://www.ssllabs.com/ssltest/

[143] Hardenize, "Hardenize." [Online]. Available: https://www.hardenize.com/

[144] Mozilla, "Cipherscan." [Online]. Available: https://github.com/mozilla/cipherscan

[145] T. Chia, "tlsenum." [Online]. Available: https://github.com/Ayrx/tlsenum

[146] A. Bezroutchko, "sslcautdit." [Online]. Available: https://github.com/abbbe/sslcaudit

[147] J. Hodges, "Hows my SSL." [Online]. Available: https://www.howsmyssl.com

[148] Qualys, "Ssl labs client test." [Online]. Available: https://clienttest.ssllabs.com:8443/ssltest/viewMyClient.html

[149] BrowserLeaks, "Browserleaks." [Online]. Available: https://browserleaks.com/ssl

[150] A. King, L. Garron, and C. Thompson, "Badssl." [Online]. Available: https://badssl.com/

[151] eco Verband der Internetwirtschaft, R.-U. Bochum, Hackmanit, and cms Garden, "SIWECOS, auf der sicheren seite." [Online]. Available: https://browserleaks.com/ssl

[152] Riku, Antti, Matti, and Mehta, "Heartbleed, CVE-2014-0160," 2015, http://heartbleed.com/.

[153] J. P. Drees, P. Gupta, E. Hüllermeier, T. Jager, A. Konze, C. Priesterjahn, A. Ramaswamy, and J. Somorovsky, "Automated detection of side channels in cryptographic protocols: DROWN the ROBOTs!" Cryptology ePrint Archive, Paper 2021/591, 2021, https://eprint.iacr.org/2021/591. [Online]. Available: https://eprint.iacr.org/2021/591

[154] R. A. Fisher, "On the interpretation of χ 2 from contingency tables, and the calculation of p," *Journal of the Royal Statistical Society*, vol. 85, no. 1, pp. 87–94, 1922.

[155] K. P. F.R.S., "On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900. [Online]. Available: https://doi.org/10.1080/14786440009463897

[156] P. Waltenberg, "Openssl security advisory," CVE-2018-0733. [Online]. Available: https://www.openssl.org/news/secadv/20180327.txt

[157] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman, "A search engine backed by Internet-wide scanning," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 542–553. [Online]. Available: https://doi.org/10.1145/2810103.2813703

[158] R. D. Graham, "MASSCAN: Mass IP port scanner." [Online]. Available: https://github.com/robertdavidgraham/masscan

[159] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast Internet-wide scanning and its security applications," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 605–620. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric

[160] W. Mayer, A. Zauner, M. Schmiedecker, and M. Huber, "No need for black chambers: Testing TLS in the e-mail ecosystem at large," *CoRR*, vol. abs/1510.08646, 2015. [Online]. Available: http://arxiv.org/abs/1510.08646

[161] V. L. Pochat, T. V. Goethem, and W. Joosen, "Evaluating the long-term effects of parameters on the characteristics of the tranco top sites ranking," in *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/cset19/presentation/lepochat

[162] Q. Scheitle, O. Hohlfeld, J. Gamba, J. Jelten, T. Zimmermann, S. D. Strowes, and N. Vallina-Rodriguez, "A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists," in *Internet Measurement Conference (IMC'18), IMC'18 Community Contribution Award*. Boston, USA: ACM, Nov. 2018, pp. 478–493.

[163] A. Popov, "Prohibiting RC4 Cipher Suites," RFC 7465 (Proposed Standard), RFC Editor, Fremont, CA, USA, Feb. 2015, updated by RFC 8996. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7465.txt

[164] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[165] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. A. Porras, "Delta: A security assessment framework for software-defined networks," in *NDSS*, 2017.

[166] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 114–129. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6956560

[167] Y. Chen and Z. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 793–804. [Online]. Available: http://stap.sjtu.edu.cn/images/c/ca/Mucert.pdf

[168] K. Kleine and D. E. Simos, "Coveringcerts: combinatorial methods for X.509 certificate testing," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 69–79.

[169] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, "Symcerts: Practical symbolic execution for exposing noncompliance in X.509 certificate validation implementations," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 503–520.

[170] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 521–538. [Online]. Available: https://www.ieee-security.org/TC/SP2017/papers/414.pdf

[171] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient domain-independent differential testing," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 615–632. [Online]. Available: http://www.cs.columbia.edu/~suman/docs/nezha.pdf

[172] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2014.2372785

[173] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews, "Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks," in *23rd USENIX Security Symposium, San Diego, USA*, August 2014.

[174] G. Vranken, "Network-emulator," 2020. [Online]. Available: https://github.com/guidovranken/network-emulator

[175] O. development team, "OpenSSL – Cryptography and SSL/TLS Toolkit," https://www.openssl.org.

[176] Rambus, "MatrixSSL. Compact Embedded SSL/TLS stack," http://www.matrixssl.org/.

[177] J. Lloyd, "Botan: Crypto and TLS for C++11," http://botan.randombit.net/.

[178] A. Smotrakov, "tlsbunny," https://github.com/artem-smotrakov/tlsbunny, 2019, accessed: August 23, 2023.

[179] H. Kario, "tlsfuzzer," 2018. [Online]. Available: https://github.com/tomato42/tlsfuzzer

[180] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[181] R. Natella and V.-T. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[182] "AddressSanitizer," 2022. [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html

[183] 2022. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[184] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[185] H. Raffelt, M. Merten, B. Steffen, and T. Margaria, "Dynamic testing via automata learning," *STTT*, vol. 11, no. 4, pp. 307–324, 2009. [Online]. Available: https://doi.org/10.1007/s10009-009-0120-7

[186] F. W. Vaandrager, "Model learning," *Commun. ACM*, vol. 60, no. 2, pp. 86–95, 2017. [Online]. Available: https://doi.org/10.1145/2967606

[187] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines—a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996. [Online]. Available: https://ieeexplore.ieee.org/document/533956

[188] C. Y. Cho, D. Babic, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS 2010. ACM, Oct. 2010, pp. 426–439. [Online]. Available: https://doi.org/10.1145/1866307.1866355

[189] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90052-6

[190] M. Isberner, F. Howar, and B. Steffen, "The TTT algorithm: A redundancy-free approach to active automata learning," in *Runtime Verification: 5th International Conference, RV 2014, Proceedings*, ser. LNCS. Springer, Sep. 2014, vol. 8734, pp. 307–322. [Online]. Available: https://doi.org/10.1007/978-3-319-11164-3_26

[191] ——, "The open-source LearnLib - A framework for active automata learning," in *Computer Aided Verification - 27th International Conference, CAV*, ser. LNCS, vol. 9206. Springer, 2015, pp. 487–495. [Online]. Available: https://dx.doi.org/10.1007/978-3-319-21690-4_32

[192] M. Isberner, "Foundations of active automata learning: An algorithmic perspective," Ph.D. dissertation, Technical University Dortmund, Germany, 2015. [Online]. Available: http://hdl.handle.net/2003/34282

[193] T. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, 1978.

[194] F. B. Khendek, "Test selection based on finite state models," *IEEE Transactions on software engineering*, vol. 17, no. 591-603, pp. 10–1109, 1991.

[195] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security 15)*.

Washington, D.C.: USENIX Association, Aug. 2015, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[196] D. Wagner and B. Schneier, "Analysis of the SSL 3.0 protocol," in *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*. Berkeley, CA, USA: USENIX Association, 1996, pp. 29–40.

[197] E. Rescorla, H. Tschofenig, and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3," RFC 9147 (Proposed Standard), RFC Editor, Fremont, CA, USA, Apr. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9147.txt

[198] V. Klíma, O. Pokorný, and T. Rosa, "Attacking RSA-based sessions in SSL/TLS," in *Cryptographic Hardware and Embedded Systems - CHES 2003*, C. D. Walter, Ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 426–440.

[199] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom, "The 9 lives of Bleichenbacher's CAT: New cache attacks on TLS implementations," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 435–452.

[200] T. Jager, J. Schwenk, and J. Somorovsky, "On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1185–1196. [Online]. Available: https://doi.org/10.1145/2810103.2813657

[201] K. Dorey, N. Chang-Fong, and A. Essex, "Indiscreet logs: Diffie-Hellman backdoors in TLS," in *NDSS*, 2017.

[202] M. Vanhoef and F. Piessens, "All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 97–112. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/vanhoef

[203] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt, "On the security of RC4 in TLS," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 305–320. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/alFardan

[204] D. Poddebniak, F. Ising, H. Böck, and S. Schinzel, "Why TLS is better without STARTTLS: A security analysis of STARTTLS in the email context," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 4365–4382. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/poddebniak

[205] S. Checkoway, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, H. Shacham, and M. Fredrikson, "On the practical exploitability of dual EC in TLS implementations," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 319–335. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/checkoway

[206] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating SSL certificates in non-browser software," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 38–49. [Online]. Available: https://doi.org/10.1145/2382196.2382204

[207] Y. Chen and Z. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 793–804. [Online]. Available: https://doi.org/10.1145/2786805.2786835

[208] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying wolf: An empirical study of SSL warning effectiveness," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. USA: USENIX Association, 2009, p. 399–416.

[209] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. USA: USENIX Association, 2003, p. 1.

[210] O. Aciiçmez, W. Schindler, and c. K. Koç, "Improving brumley and boneh timing attack on unprotected SSL implementations," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 139–146. [Online]. Available: https://doi.org/10.1145/1102120.1102140

[211] B. Brumley and N. Tuveri, "Remote Timing Attacks Are Still Practical," in *Computer Security - ESORICS 2011*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Sep. 2011, vol. 6879.

[212] Z. Song and S. Qing, "Applying NCP logic to the analysis of SSL 3.0," in *Information and Communications Security*, S. Qing, T. Okamoto, and J. Zhou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 155–166.

[213] J. C. Mitchell, V. Shmatikov, and U. Stern, "Finite-state analysis of SSL 3.0," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. USA: USENIX Association, 1998, p. 16.

[214] Y. Cheng, W. Kang, and M. Xiao, "Model checking of SSL 3.0 protocol based on spin," in *2010 2nd International Conference on Industrial and Information Systems*, vol. 2, 2010, pp. 401–403.

[215] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the security of TLS-DHE in the standard model," in *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 273–293.

[216] H. Krawczyk, K. G. Paterson, and H. Wee, "On the security of the TLS protocol: A systematic analysis," in *Advances in Cryptology – CRYPTO 2013*, R. Canetti and J. A. Garay, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 429–448.

[217] J. Jonsson and B. S. Kaliski, "On the security of RSA encryption in TLS," in *Advances in Cryptology — CRYPTO 2002*, M. Yung, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 127–142.

[218] C. Brzuska, H. Jacobsen, and D. Stebila, "Safely exporting keys from secure channels," in *Advances in Cryptology – EUROCRYPT 2016*, M. Fischlin and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 670–698.

[219] K. G. Paterson, T. Ristenpart, and T. Shrimpton, "Tag size does matter: Attacks and proofs for the TLS record protocol," in *Advances in Cryptology – ASIACRYPT 2011*, D. H. Lee and X. Wang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 372–389.

[220] Y. Li, S. Schäge, Z. Yang, F. Kohlar, and J. Schwenk, "On the security of the pre-shared key ciphersuites of TLS," in *Public-Key Cryptography – PKC 2014*, H. Krawczyk, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 669–684.

[221] F. Giesen, F. Kohlar, and D. Stebila, "On the security of TLS renegotiation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 387–398. [Online]. Available: https://doi.org/10.1145/2508859.2516694

[222] P. Morrissey, N. P. Smart, and B. Warinschi, "A modular security analysis of the TLS handshake protocol," in *Advances in Cryptology - ASIACRYPT 2008*, J. Pieprzyk, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 55–73.

[223] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "EverParse: Verified secure Zero-Copy parsers for authenticated message formats," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1465–1482. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

[224] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with verified cryptographic security," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 445–459.

[225] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 483–502.

[226] J. Brendel, M. Fischlin, and F. Günther, "Breakdown resilience of key exchange protocols: NewHope, TLS 1.3, and hybrids," in *Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 521–541. [Online]. Available: https://doi.org/10.1007/978-3-030-29962-0_25

[227] C. Badertscher, C. Matt, U. Maurer, P. Rogaway, and B. Tackmann, "Augmented secure channels and the goal of the TLS 1.3 record layer," in *Proceedings of the 9th International Conference on Provable Security - Volume 9451*, ser. ProvSec 2015. Berlin, Heidelberg: Springer-Verlag, 2015, p. 85–104. [Online]. Available: https://doi.org/10.1007/978-3-319-26059-4_5

[228] M. Bellare and B. Tackmann, "The multi-user security of authenticated encryption: AES-GCM in TLS 1.3," in *Advances in Cryptology – CRYPTO 2016*, M. Robshaw and J. Katz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 247–276.

[229] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of TLS 1.3," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1773–1788. [Online]. Available: https://doi.org/10.1145/3133956.3134063

[230] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 470–485.

[231] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the TLS 1.3 record layer," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 463–482.

[232] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol," Cryptology ePrint Archive, Paper 2016/081, 2016, https://eprint.iacr.org/2016/081. [Online]. Available: https://eprint.iacr.org/2016/081

[233] M. Fischlin and F. Günther, "Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 60–75.

[234] M. Fischlin, F. Günther, B. Schmidt, and B. Warinschi, "Key confirmation in key exchange: A formal treatment and implications for TLS 1.3," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 452–469.

[235] M. Kohlweiss, U. Maurer, C. Onete, B. Tackmann, and D. Venturi, "(de-)constructing TLS 1.3," in *INDOCRYPT*, 2015.

[236] H. Krawczyk and H. Wee, "The OPTLS protocol and TLS 1.3," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 81–96.

[237] C. Patton and T. Shrimpton, "Partially specified channels: The TLS 1.3 record layer without elision," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1415–1428. [Online]. Available: https://doi.org/10.1145/3243734.3243789

[238] H. Davis, D. Diemert, F. Günther, and T. Jager, "On the concrete security of TLS 1.3 PSK mode," in *Advances in Cryptology – EUROCRYPT 2022*, O. Dunkelman and S. Dziembowski, Eds. Cham: Springer International Publishing, 2022, pp. 876–906.

[239] N. Aviram, K. Gellert, and T. Jager, "Session resumption protocols and efficient forward security for TLS 1.3 0-RTT," *J. Cryptol.*, vol. 34, no. 3, jul 2021. [Online]. Available: https://doi.org/10.1007/s00145-021-09385-0

[240] J. Bozic, L. Marsso, R. Mateescu, and F. Wotawa, "A Formal TLS Handshake Model in LNT," in *MARS/VPT 2018 - 3nd Workshop on Models for Formal Analysis of Real Systems and 6th International Workshop on Verification and Program Transformation*, vol. 268, Thessaloniki, Greece, Apr. 2018, pp. 1–40. [Online]. Available: https://hal.inria.fr/hal-01779151

[241] D. E. Simos, J. Bozic, F. Duan, B. Garn, K. Kleine, Y. Lei, and F. Wotawa, "Testing TLS using combinatorial methods and execution framework," in *IFIP International Conference on Testing Software and Systems*. Springer, 2017, pp. 162–177.

[242] A. Walz and A. Sikora, "Exploiting dissent: Towards fuzzing-based differential black box testing of TLS implementations," *IEEE Transactions on Dependable and Secure Computing*, 2017. [Online]. Available: https://ivesk.hs-offenburg.de/fileadmin/Einrichtungen/ivesk/files/preprint_TLS-Diff-Fuzzing_IEEE-TDSC.pdf

[243] P. Tsankov, M. T. Dashti, and D. Basin, "Secfuzz: Fuzz-testing security protocols," in *Automation of Software Test (AST), 2012 7th*

International Workshop on. IEEE, 2012, pp. 1–7. [Online]. Available: https://www.inf.ethz.ch/personal/basin/pubs/ast12.pdf

[244] T. Pornin, "Boarssl," 2017. [Online]. Available: https://www.bearssl.org/boarssl.html

[245] H. Asadian, P. Fiterau-Brostean, B. Jonsson, and K. Sagonas, "Applying symbolic execution to test implementations of a network protocol against its specification," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2022, pp. 70–81.

[246] N. van Drueten, "Security analysis of DTLS 1.2 implementations," Bachelor thesis, Radboud University, Nijmegen, The Netherlands, 2019.

[247] C. McMahon Stone, T. Chothia, and J. de Ruiter, "Extending automated protocol state learning for the 802.11 4-way handshake," in *Computer Security*, ser. LNCS, vol. 11098. Cham: Springer International Publishing, Aug. 2018, pp. 325–345.

[248] G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter, "Automated reverse engineering using Lego®," in *8th USENIX Workshop on Offensive Technologies*, ser. WOOT 14. USENIX Association, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/chalupar

[249] D. Adrian, Z. Durumeric, G. Singh, and J. A. Halderman, "Zippier ZMap: Internet-Wide scanning at 10 gbps," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/adrian

[250] L. Izhikevich, R. Teixeira, and Z. Durumeric, "LZR: Identifying unexpected internet services," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3111–3128. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/izhikevich

[251] D. Kumar, Z. Wang, M. Hyder, J. Dickinson, G. Beck, D. Adrian, J. Mason, Z. Durumeric, J. A. Halderman, and M. Bailey, "Tracking certificate misissuance in the wild," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 785–798.

[252] B. VanderSloot, J. Amann, M. Bernhard, Z. Durumeric, M. Bailey, and J. A. Halderman, "Towards a complete view of the certificate ecosystem," in *Proceedings of the 2016 Internet Measurement Conference*, ser. IMC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 543–549. [Online]. Available: https://doi.org/10.1145/2987443.2987462

[253] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the HTTPS certificate ecosystem," in *Proceedings of the 2013 conference on Internet measurement conference*, 2013, pp. 291–304.

[254] G. Wan, L. Izhikevich, D. Adrian, K. Yoshioka, R. Holz, C. Rossow, and Z. Durumeric, "On the origin of scanning: The impact of location on Internet-wide scans," in *Proceedings of the ACM Internet Measurement Conference*, ser. IMC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 662–679. [Online]. Available: https://doi.org/10.1145/3419394.3424214

[255] C. Simoiu, W. Nguyen, and Z. Durumeric, "An Empirical Analysis of HTTPS Configuration Security," *arXiv e-prints*, p. arXiv:2111.00703, Nov. 2021.

[256] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your ps and qs: Detection of widespread weak keys in network devices," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 205–220. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger

[257] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 conference on internet measurement conference*. ACM, 2014, pp. 475–488.

[258] W. Mayer, A. Zauner, M. Schmiedecker, and M. Huber, "No need for black chambers: Testing TLS in the e-mail ecosystem at large," *CoRR*, vol. abs/1510.08646, 2015. [Online]. Available: http://arxiv.org/abs/1510.08646

[259] L. Valenta, N. Sullivan, A. Sanso, and N. Heninger, "In search of curveswap: Measuring elliptic curve implementations in the wild," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 384–398.

[260] P. Kotzias, A. Razaghpanah, J. Amann, K. G. Paterson, N. Vallina-Rodriguez, and J. Caballero, "Coming of age: A longitudinal study of TLS deployment," in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 415–428. [Online]. Available: https://doi.org/10.1145/3278532.3278568

[261] P. Schwabe, D. Stebila, and T. Wiggers, "More efficient post-quantum KEMTLS with pre-distributed public keys," in *European Symposium on Research in Computer Security*. Springer, 2021, pp. 3–22.

[262] ——, "Post-quantum TLS without handshake signatures," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1461–1480.

[263] C. Paquin, D. Stebila, and G. Tamvada, "Benchmarking post-quantum cryptography in TLS," in *Post-Quantum Cryptography*, J. Ding and J.-P. Tillich, Eds. Cham: Springer International Publishing, 2020, pp. 72–91.

[264] E. Crockett, C. Paquin, and D. Stebila, "Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH," *Cryptology ePrint Archive*, 2019.

[265] N. Howgrave-Graham and N. P. Smart, "Lattice attacks on digital signature schemes," *Des. Codes Cryptogr.*, vol. 23, no. 3, pp. 283–290, 2001.

[266] P. Q. Nguyen and I. E. Shparlinski, "The insecurity of the elliptic curve digital signature algorithm with partially known nonces," *Des. Codes Cryptogr.*, vol. 30, no. 2, pp. 201–217, 2003. [Online]. Available: https://doi.org/10.1023/A:1025436905711

[267] J. Kelsey, B. Schneier, and D. Wagner, "Protocol interactions and the chosen protocol attack," in *Security Protocols*. Springer Berlin Heidelberg, 1998, pp. 91–104. [Online]. Available: https://doi.org/10.1007%2Fbfb0028162

[268] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel, "A Cross-protocol Attack on the TLS Protocol," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 62–72. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382206

[269] N. Drucker and S. Gueron, "Selfie: reflections on TLS 1.3 with PSK," Cryptology ePrint Archive, Report 2019/347, 2019, https://eprint.iacr.org/2019/347.

[270] J. Topf, "The HTML form protocol attack," 2001, published on the Bugtraq mailing list on 2001-08-15. https://www.jochentopf.com/hfpa/hfpa.pdf (accessed 2019-10-18). [Online]. Available: https://www.jochentopf.com/hfpa/hfpa.pdf

[271] W. Alcorn, "Inter-protocol communication," 2006, https://web.archive.org/web/20111229080404/http://www.bindshell.net/papers/ipc.html (accessed 2019-06-26). [Online]. Available: https://web.archive.org/web/20111229080404/http://www.bindshell.net/papers/ipc.html

[272] ——, "Inter-process exploitation," 2007, https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/inter-protocol_exploitation.pdf (accessed 2019-06-26). [Online]. Available: https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/inter-protocol_exploitation.pdf

[273] Gnucitizen.org, "Hacking the interwebs," 2008, https://www.gnucitizen.org/blog/hacking-the-interwebs/ (accessed 2019-07-09). [Online]. Available: https://www.gnucitizen.org/blog/hacking-the-interwebs/

[274] A. Quina, "Inter-protocol communication - exploitation," 2012, https://www.secforce.com/blog/2012/11/inter-protocol-communication/ (accessed 2019-07-09). [Online]. Available: https://www.secforce.com/blog/2012/11/inter-protocol-communication/

[275] M. Orrù, "Revitalizing the Inter-Protocol Exploitation with BeEF Bind," 2012, https://blog.beefproject.com/2012/11/revitalizing-inter-protocol.html (accessed 2019-07-09). [Online]. Available: https://blog.beefproject.com/2012/11/revitalizing-inter-protocol.html

[276] N. Grgoire, "Trying to hack Redis via HTTP requests," 2014, https://www.agarri.fr/blog/archives/2014/09/11/trying_to_hack_redis_via_http_requests/index.html (accessed 2019-10-14). [Online]. Available: https://www.agarri.fr/blog/archives/2014/09/11/trying_to_hack_redis_via_http_requests/index.html

[277] A. Weaver, "Cross-site printing," 2007, http://web.archive.org/web/20090919174421/http://www.net-security.org/dl/articles/CrossSitePrinting.pdf.

[278] R. Hansen, "Javascript spam," 2007, http://web.archive.org/web/20090913204859/http://ha.ckers.org/blog/20070325/javascript-spam.

[279] T. Prynn, "Cross-protocol request forgery," 2018, https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2018/cprf-1.pdf (accessed 2019-10-14). [Online]. Available: https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2018/cprf-1.pdf

[280] M. Belshe, R. Peon, and M. Thomson (Ed.), "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540 (Proposed Standard), RFC Editor, Fremont, CA, USA, May 2015, obsoleted by RFC 9113, updated by RFC 8740. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7540.txt

[281] S. Gauci, "Extended HTML Form Attack," 2002, https://eyeonsecurity.org/papers/extendedformattack.html (accessed 2019-10-18). [Online]. Available: https://eyeonsecurity.org/papers/extendedformattack.html

[282] ——, "The Extended HTML Form attack revisited," 2008, https://dl.packetstormsecurity.net/papers/web/the-extended-html-form-attack-revisited.pdf (accessed 2020-09-26). [Online]. Available: https://dl.packetstormsecurity.net/papers/web/the-extended-html-form-attack-revisited.pdf

[283] J. Horn, "Two cross-protocol MitM attacks on browsers," 2015, https://var.thejh.net/http_ftp_cross_protocol_mitm_attacks.pdf (accessed 2020-08-27). [Online]. Available: https://var.thejh.net/http_ftp_cross_protocol_mitm_attacks.pdf

[284] J. Müller, V. Mladenov, J. Somorovsky, and J. Schwenk, "SoK: Exploiting network printers," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 213–230.

[285] M. Zalewski, "American fuzzy lop," 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[286] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ : Combining incremental steps of fuzzing research," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[287] "honggfuzz," 2018. [Online]. Available: http://honggfuzz.com

[288] S. Frankel and S. Krishnan, "IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap," RFC 6071 (Informational), RFC Editor, Fremont, CA, USA, Feb. 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6071.txt

[289] P. Fiterău-Broştean, R. Janssen, and F. W. Vaandrager, "Combining model learning and model checking to analyze TCP implementations," in *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part II*, ser. LNCS, vol. 9780. Springer, 2016, pp. 454–471. [Online]. Available: https://doi.org/10.1007/978-3-319-41540-6_25

[290] P. Fiterău-Broştean, T. Lenaerts, J. de Ruiter, E. Poll, F. W. Vaandrager, and P. Verleg, "Model learning and model checking of SSH implementations," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. ACM, 2017, pp. 142–151. [Online]. Available: https://doi.org/10.1145/3092282.3092289