Security Testing of Embedded TLS Implementations

Angel Yafte Lomeli Ramos

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 01.07.2022

Supervisor

Prof Jan-Erik Ekberg and Prof Danilo Gligoroski

Advisor

MSc Arto Niemi

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Angel Yafte Lomeli Ramos

Title Security Testing of Embedded TLS Implementations

Degree programme Master's Programme in Security and Cloud Computing

Major Security and Cloud Computing **Code of major** SCI3113

Supervisor Prof Jan-Erik Ekberg and Prof Danilo Gligoroski

Advisor MSc Arto Niemi

Date 01.07.2022 **Number of pages** 99 **Language** English

Abstract

The Transport Layer Security (TLS) protocol is by far the most popular cryptographic protocol used to secure data being exchanged on the Internet. The latest version, TLS 1.3, provides a set of security properties on top of what was already included in previous versions, such as channel binding, downgrade protection, and non-replayability, along with improved performance and exclusion of less secure algorithms that were used before. However, vulnerabilities are constantly found and reported in implementations of TLS, especially those developed specifically for embedded devices, as they require code optimizations that sometimes leave necessary checks out of the picture, giving rise to security flaws.

To improve the quality and security of these embedded implementations, it is recommended to apply software testing throughout the development process. Fuzz testing, or simply fuzzing, is an effective testing technique that has been successfully used in the past to find bugs and vulnerabilities in different kinds of applications. In fuzzing tests, semi-valid test cases are generated randomly either by modifying valid seeds or by following a specification or model and fed as input to the target program, while monitoring its behavior.

Unfortunately, most existing fuzzing tools are focused on file-based or standard input applications, and they are not very effective for testing cryptographic protocols where input messages are subject to integrity checks and need to be encrypted and decrypted constantly. Additionally, to the best of our knowledge, there is a lack of a comprehensive tool or set of guidelines that specify how to test embedded TLS implementations.

In this thesis, we develop a testing framework that can be used to measure the security of embedded TLS implementations by combining fuzzing techniques with hand-crafted test cases. We then use this framework to test HTLS, a TLS library developed by Huawei.

Keywords Security testing, Vulnerability Analysis , Fuzzing , Embedded Systems

Acknowledgments

I want to thank my supervisors Jan-Erik Ekberg and Danilo Gligoroski for their advice, and special thanks to my advisor Arto Niemi for his guidance throughout this research.

I would also like to thank my family for their support and most of all I thank Fernanda Silva for being by my side during this journey.

Otaniemi, 01.07.2022

Angel Yafte Lomeli Ramos

With the support of the Erasmus+ Programme of the European Union

Contents

Acronyms and abbreviations

0-RTT	0 Round-Trip Time
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
AFL	American Fuzzy Lop
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
CCA	Chosen-Ciphertext Attack
CVSS	Common Vulnerability Scoring System
DER	Distinguished Encoding Rules
DES	Data Encryption Standard
DH	Diffie-Hellman Key Exchange
DHE	Ephemeral Diffie-Hellman Key Exchange
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
DSE	Dynamic Symbolic Execution
DSS	Digital Signature Standard
DTLS	Datagram Transport Layer Security
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDHE	Ephemeral Diffie-Hellman over Elliptic Curves
ECDSA	Elliptic Curve Digital Signature Algorithm (ECDSA)
HKDF	HMAC-based Extract-and-Expand Key Derivation Function
GMT	Generic Message Tree
HSSL	Helsinki System Security Laboratory
IETF	Internet Engineering Task Force
IoT	Internet of Things
KDF	Key Derivation Function
MAC	Message Authentication Code
MitM	Man-in-the-Middle
NIST	National Institute of Standards and Technology
OSI	Open Systems Interconnection
PKCS	Public-Key Cryptography Standards
PSK	Pre-Shared Key
PUT	Program Under Test
RFC	Request For Comments
RNG	Random Number Generator
SHS	Secure Hash Standard
SPKI	Subject Public Key Info
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TLV	Tag-Length-Value
TPL	TLS Presentation Language
TSL	Trusted Sockets Layer
UDP	User Datagram Protocol

1 Introduction

By the end of 2021, there were about 11.3 billion Internet-connected devices in the world [1]. This number is expected to increase exponentially in coming years due mainly to the popularity of Internet-of-Things (IoT) devices, which are constantly being adopted for day to day use. These devices exchange huge amounts of data with one another, which brings the need for cryptographic protocols that can be used to secure this data.

Transport Layer Security (TLS) [2] is the most widely used cryptographic protocol for securing data communications [3]. It was designed to fulfill security requirements including endpoint authentication, confidentiality, and integrity protection. Unfortunately, previous versions of TLS have been shown to be vulnerable to different kinds of attacks that break the inner mechanisms of the protocol and allow adversaries to read or modify the transported data in malicious ways. Such vulnerabilities have been in both the protocol design and its implementations. For example, the Heartbleed vulnerability [4], discovered in 2014, allowed an attacker to retrieve large amounts of bytes from the server's memory, even sensitive information, such as usernames, passwords, or cryptographic keys. The Heartbleed vulnerability was present in the OpenSSL implementation of the Heartbeat extension for TLS [5]. The extension was developed for the purpose of solving renegotiation and authentication problems in TLS sessions.

For embedded systems, such as those controlling IoT devices, C is still the predominant programming language. It provides low-level memory access and has overall better performance for systems with constrained resources. Other advantages of choosing the C language for embedded systems include [6]:

1. It is available and can be compiled for most processor architectures.
2. It allows fine-grained control of memory via pointers, which eases the programming of hardware components.
3. C statements are well mapped to machine operations, making it faster to execute than other languages.
4. It has low runtime and smaller memory footprint when compared to others, such as C++. Compiled binary files tend to have smaller sizes.
5. Most drivers and kernels are also developed in C, including the Linux kernel [7], giving little motivation to choose a new language.

The C programming language, however, has been known to be hard to use securely [8, pp. 21-25]. Often, programmers fail to implement logical security controls assuming that unexpected behaviors are inherently handled by the language or compiler, which is not the case. This lack of controls gives rise to security holes that can be abused to cause great damage. C compilers are unable (and not required) to diagnose undefined behaviors during compilation. This, combined with compiler optimizations, allows for erroneous instruction blocks to be written by programmers

and remain undetected until it is too late. Bugs such as writing beyond buffer boundaries, missing checks for integer overflows, and function calls with wrong arguments or conventions are not seen at compile time and thus, unreported.

Pointer operations are powerful for control of inputs and outputs or hardware peripherals, but often lead to confusion. In combination with the lack of type safety in the C language, the risk of missing potential vulnerabilities increases considerably. Type safety [9] expects that after applying an operation to an argument of a specific type, the result will have the same type. In C, however, values can have several interpretations and their types are often converted using, e.g., type casting [8, pp. 246-256]. Pointers often need to be converted before use and integers, for example, are capable of being converted implicitly from signed to unsigned or vice versa depending on the context. These variations in types are also common root causes of security flaws.

In order to reduce the possibility of adding unwanted software bugs in applications, developers use a combination of manual and automatic testing tools and techniques [10]. Software testing helps not only in finding bugs, but also in ensuring that the tested program complies with its specific quality requirements. One of the most popular software testing methods is fuzz testing, also simply called fuzzing [11], which refers to the process of using semi-random inputs to test an application. Researchers have successfully used fuzzing and other software testing methodologies to find bugs in TLS implementations [12, 13, 14]. However, to the best of our knowledge, there does not exist a comprehensive testing framework in the literature for finding errors and measuring the security of embedded TLS 1.3 implementations.

The aim of this thesis is to study the TLS protocol security and to develop a testing framework that targets implementations of the latest TLS version, TLS 1.3, with special focus on embedded systems and resource-constrained devices. This testing framework will be used to test HTLS, a lightweight TLS 1.3 library developed by the Helsinki System Security Laboratory (HSSL) from Huawei Technologies Oy in Finland. HSSL is a research lab that focuses on system and device security, mainly, hardware-backed security and confidential computing (e.g., TrustZone-style Trusted Security Environments and secure enclaves [15]), compile-time protection (e.g., pointer authentication), runtime protection (e.g., kernel integrity [16]), key storage, PKI, cryptography, IoT device management, among others. One of the goals of HSSL was to make HTLS ready for Open Source release. There was a need for an extensive defect/fuzz testing process as the one performed in this research to allow HTLS to transition from a research prototype to an Open Source library ready for production.

The rest of this thesis is organized as follows: Section 2 provides background information on the related cryptographic algorithms and concepts, the TLS protocol, some vulnerabilities found in previous versions and how TLS 1.3 mitigates them, as well as introducing fuzzing as a method to find bugs. Section 3 provides an overview of HTLS, the TLS implementation that will be tested. Section 4 details the design of the automated testing framework, explains the selection of tools and techniques that were included and introduces *eGMT*, an extended fuzzing tool we designed to fuzz TLS 1.3 implementations. Section 5 presents the results of the tests performed

with the testing framework including the found bugs and their fixes. Finally, Section 6 concludes the thesis with closing remarks and future work.

The fuzzer introduced in this thesis is also described in the paper *eGMT: Deep Fuzzing of Cryptographic Protocols Using Syntax Tree Mutation*, which is in the process of being submitted by HSSL.

For this research, I did all the coding of the fuzzer, eGMT, and wrote a test suite that uses this fuzzer along with static test cases from the Wycheproof project [17]. I designed the new eGMT operators, implemented the improvements on previous operators as discussed in Section 4, and wrote around half of the current draft of the paper for eGMT. The HSSL team wrote the fixes for the vulnerabilities found in the target system, comprised of both HTLS and Tinycrypt [18], a low-level cryptographic library used by HTLS.

2 Background

This section discusses cryptography background, the TLS protocol and fuzzing. This information is required to understand the security issues found in the target implementation, discussed in Section 5, and how the testing framework works. It will also serve as a reference for the security issues that were discovered in previous versions of TLS and the controls included in the latest version to remediate them.

The main goal of the TLS protocol is to provide means for creating a secure channel between a client and server. A secure channel [19, p. 7] is defined as a path for data transfer between two entities that fulfills the following properties:

End-point Authentication. The channel should always provide secure and reliable authentication of the server side. The client side can also be authenticated but this is optional in TLS.

Confidentiality. The data being sent through the secure channel should be readable only by the client and server. Optionally, in TLS, both endpoints can also add padding of the messages to obscure their lengths (to mitigate traffic analysis). Length obscurity is not done by default in TLS.

Integrity. Message modification by an attacker should be detected reliably to protect integrity.

Order protection/non-replayability. The protocol should not allow a party to receive a record it already accepted or a future record if it has not processed the corresponding previous records.

Some of the other security properties provided by the latest version, TLS 1.3, are:

Key confirmation. Both ends of the communication should have the same session keys when the handshake is complete. A cryptographic verification allows each party to confirm that the key they computed is correct and equal to what the other party computed. This is achieved in TLS 1.3 via the `Finished` message.

Binding of authentication handshake and session keys. Channel binding [20] is a protection mechanism against Man-in-the-Middle (MitM) attacks that ensures that the entity that was authenticated is the same as the one that obtains access to the session protection keys. This is particularly useful when authentication happens at a different layer than key negotiation. TLS 1.3 solves this by including the `Certificate` and `CertificateVerify` messages (used for endpoint authentication) in the handshake transcript, which is required for session key derivation. In addition, the `CertificateVerify` message is valid for each specific handshake since it also uses the transcript, thus an attacker cannot reuse it in other sessions.

Downgrade protection. All cryptographic parameters for both parties should be the same even if there was an active attacker in the network.

Forward secrecy in relation to long-term keys. Compromise of the long-term keying material shall not imply compromise of the session keys or previous communications. This is not achieved when a Pre-Shared Key (PSK) is used in PSK Key Exchange Mode.

These properties should hold true even in the presence of an attacker with full control of the network [2], such as a Dolev-Yao attacker [21]. It is also expected that each of the endpoints involved in the communication are secure so that, e.g., attackers on the same machine cannot steal the keys. To ensure that the secure channel has these properties, the algorithms discussed below are used.

2.1 Cryptographic Algorithms and Notations

Before talking about the TLS protocol, it is important to explain some of the cryptographic algorithms it uses to better understand the test target and environment that will be shown later, as well as the nature of some of the tests that were implemented and the vulnerabilities that were found, discussed in Section 5. This section focuses solely on the algorithms and cipher suites used by TLS 1.3, as this is the version that will be tested. This is not meant to be an extensive explanation of each algorithm, but rather an overview of the details that need to be understood.

2.1.1 SHA

The Secure Hash Algorithm (SHA) [22] is a set of cryptographic hash functions used to generate a unique value often called a "digest" out of any message. The SHA family consists of a flawed and deprecated SHA-0 algorithm, the SHA-1 algorithm with an output of 160 bit length, and the SHA-2 algorithms [23, 234-236]. SHA-2 is another subgroup of SHA which consists of the algorithms SHA-224, SHA-256, SHA-384, and SHA-512, where the number portion of the name corresponds to the bit length of their outputs. An additional algorithm called SHA-3 [24] has also been adopted. This was a result of the NIST cryptographic hash function competition launched in 2007 [25]. SHA-3 is based on an algorithm called *Keccack* and has several differences in comparison to its predecessors (e.g., it is based on *unkeyed* permutations, it does not use the Merkle-Damgård transform, etc). It is also standardized by NIST and meant to serve as a replacement for SHA-2 if significant weaknesses are discovered.

SHA-1 had been known for years to be a weak algorithm, as theoretical attacks and analyses had been published by researchers since at least 2005 [26]. In 2017, the theories were proven correct when a collision for two different documents was found [27]. Even though NIST had deprecated the algorithm in 2011 due to the theoretical weaknesses, it was still being widely used when the collision was found, especially for TLS certificate signatures. This further motivated companies worldwide to shift to

the SHA-2 and SHA-3 algorithms instead, for which, at the time of this writing, no significant threats have been found.

The inner-workings of the SHA algorithms will not be discussed here. TLS 1.3 currently supports the SHA-2 algorithms, specifically SHA-256 and SHA-384. Any messages using SHA-1 in the handshake or for certificates should be immediately rejected, as specified in the RFC [2].

2.1.2 AES

The Advanced Encryption Standard (AES) [28] resulted from a competition launched by the National Institute of Standards and Technology (NIST) in 1997 [29]. The competition had the goal of finding a new block cipher to standardize and to replace the Data Encryption Standard (DES), which had several flaws. Three years after the competition was launched, NIST announced that the winner was the Rjindael algorithm, which was used as a foundation for what we now know as AES [23, pp. 223-225]. The AES algorithm has been subjected to thorough cryptanalysis for years and is still the standard block cipher algorithm recommended for use in symmetric cryptography.

AES uses four stages of permutations and substitutions that are performed in rounds over the plaintext and the key. It supports keys with lengths of 128, 192 or 256 bits, which also dictate the amount of rounds to perform on each block. Each block in AES has a length of 128 bits of 16 bytes, and a state is maintained throughout all rounds composed as a table of 4 by 4 bytes. This state is updated at the end of every stage, where the result is taken as the new state value. The initial value of the state is simply the block of plaintext that will be encrypted. The four stages that are applied each round to each plaintext block in their respective order are the following:

AddRoundKey. A "key expansion" routine is applied to the master key to create a key stream of the same length as the state. The state is then XORed with this key stream.

SubBytes. A substitution table called $S - Box$ is used to substitute each byte in the state independently. This $S - Box$ has a fixed value and is invertible, i.e., using the inverted version of the table to substitute the state will make it revert to its original value.

ShiftRows. A cyclic rotation to the left is done on each row of the state table depending on the position of the row. The first row does not rotate, the second row rotates one place, the third row rotates two places, and the fourth row rotates three places.

MixColumns. A transformation is applied on the state column by column, treating each column as a four-term polynomial, which makes this transformation a matrix multiplication. This transformation can be reversible. This stage is replaced by the **AddRoundKey** stage in the final round.

For decryption, the inverse version of each stage (except for **AddRoundKey**, which stays the same) is applied in reverse order to get the plaintext from an encrypted ciphertext. When the length of the plaintext is not divisible over 128 bits, a padding method must be applied so it can be divided in 128-bit blocks as required by AES.

2.1.3 DH

The Diffie-Hellman Key Exchange (DH) algorithm over finite fields [30] has been around for years, since 1976, and is considered a pillar of modern cryptography as it was one of the first public key algorithms ever created[1]. The original goal of this algorithm was to allow two parties to securely exchange cryptographic keys over an insecure channel in a way that, even if the communication was eavesdropped by an unauthorized party, the key could not be revealed.

The DH algorithm relies on the difficulty of the discrete logarithm problem (DLP) [31, pp. 261-263]. The DLP problem is based on the fact that the computation of a modular exponentiation is simple but the inverse is hard. For example:

$$a = g^x \mod n$$

where g is a primitive root of n. Then, it is said that x is the discrete logarithm of a with respect to the base g modulo n. However, the inverse calculation governed by the formula:

$$x = log_g a \mod n$$

is computationally hard, especially for very large numbers. This problem has been widely studied and researchers have found ways to reduce the complexity of the problem under specific circumstances (e.g., [34]), but no general method is known for efficiently computing a solution, at the time of this writing. The DLP is still considered to have no efficient solution under carefully chosen groups and is the pillar of several cryptographic applications [35].

DH works as follows:

Alice and Bob want to exchange a key to protect their communication, so they agree on a cyclic group G of order n and generator g on G. Both n and g can be public. Then,

1. Both Alice and Bob select the integers $a, b \in G$ respectively and keep them secret

2. Alice and Bob calculate separately their public values $A = a * g$ and $B = b * g$. Alice sends A to Bob, and Bob sends B to Alice over the network

[1] DH had been considered by many (including Bruce Schneier) the first public key algorithm ever developed [31, pp. 513-516]. However, the declassification of secret documents by the UK Government Communications Headquarters (GCHQ) in 1997 showed that the concept of public key cryptography was independently conceived by cryptographer James H. Ellis in 1970 [32], giving rise to the creation of an algorithm equivalent to what is now known as the RSA algorithm in 1973 by his colleague Clifford Cocks [33]

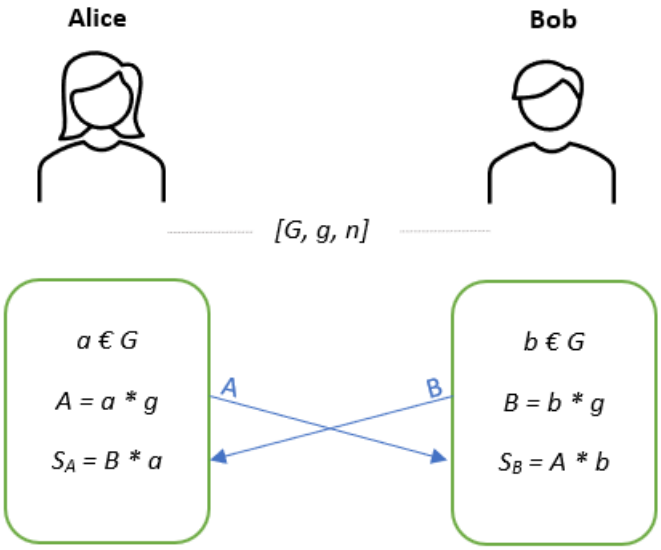

Figure 1: Diffie-Hellman key exchange protocol

3. Alice computes $S_A = a * B$ and Bob computes $S_B = b * A$
4. Since $S_A = a * B = a * b * g = b * A = S_B$, then it follows that S_A and S_B are equal and can be used as a shared secret between both parties

where all multiplications are done modulo n. This process is illustrated in figure 1. Even if an attacker is spying on the communication, it is computationally infeasible for him to recover either S_A or S_B without knowledge of either of the secret values a or b. However, this algorithm is vulnerable to Man-in-the-Middle (MitM) attacks, since the lack of authentication allows an attacker to intercept the messages and pose as both Alice and Bob to trick each of them into exchanging keys with him instead. Therefore, DH should always be combined with endpoint authentication.

2.1.4 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) was introduced in the mid-1980s, although it had not been widely adopted in cryptographic implementations until years later [2] [37, pp. 239-257]. It generally provides the same level of security as other algorithms, such as RSA, but with performance benefits, since it requires shorter keys and computations can generally be done faster. As others, ECC algorithms are based on the DLP (considered unbreakable under carefully chosen curves [38]) but the mathematical operations consist of point additions over an elliptic curve and tend to be more complex.

An elliptic curve is defined as the set of all (x, y) points with $x, y \in F_q$, where F_q is a finite field with $q \neq 2, 3$, that fulfills the *short Weierstrass equation*

[2] For example, the RFC that specified ECC-based ciphersuites for TLS was published in 2006 [36]

$$y^2 = x^3 + ax + b \mod p$$

where $a, b \in F_q$, and that satisfies the condition

$$4a^3 + 27b^2 \neq 0 \mod p$$

together with an imaginary point at infinity ∞. Then, $E(F_q)$ is the set of solutions (x, y) and n is the number of points in $E(F_q)$, also called the *order* of the curve. The curve must not have any intersections or vertices, which is true when the condition is satisfied since the discriminant $-16(4a^3 + 27b^2)$ is not zero.

The **addition** in elliptic curves is defined as the computation of the coordinates of a point $P + Q = (x_3, y_3)$ given two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, such that

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

This computation has two cases: when $P = Q$ and when $P \neq Q$. If $P \neq Q$, a third point R can be obtained by looking at the intersection in the curve after drawing a line between P and Q. R is then mirrored along the x-axis to obtain $P + Q$, as shown in Figure 2. On the other hand, if $P = Q$, a tangent line is drawn in P to obtain a second intersecting point R, which is then mirrored along the x-axis to obtain $P + Q = 2P$. This last operation is called *point doubling*.

These additions can be expressed in the equations

$$\begin{aligned} x_3 &= s^2 - x_1 - x_2 \mod p \\ y_3 &= s(x_1 - x_3) - y_1 \mod p \end{aligned}$$

where

$$s = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} \mod p; & \text{if } P \neq Q \text{ (point addition)} \\ \frac{3x_1^2 + a}{2y_1} \mod p; & \text{if } P = Q \text{ (point doubling)} \end{cases}$$

Also, the point at infinity ∞ is taken as the identity and satisfies the equations

$$\begin{aligned} P + \infty &= P \\ P + (-P) &= \infty \end{aligned}$$

where $-P$ is the inverse of P defined as the reflection of P over the x-axis, so if $P = (x_P, y_P)$, then $-P = (x_P, -y_P)$. Putting it all together, the aforementioned elements make $E(F_q)$ an Abelian group with the following properties:

- Closure: if P and Q are on the curve, it is implied that $P + Q$ is on the curve
- Inverse: each point P has an inverse, denoted $-P$
- Commutativity: $P + Q = Q + P$

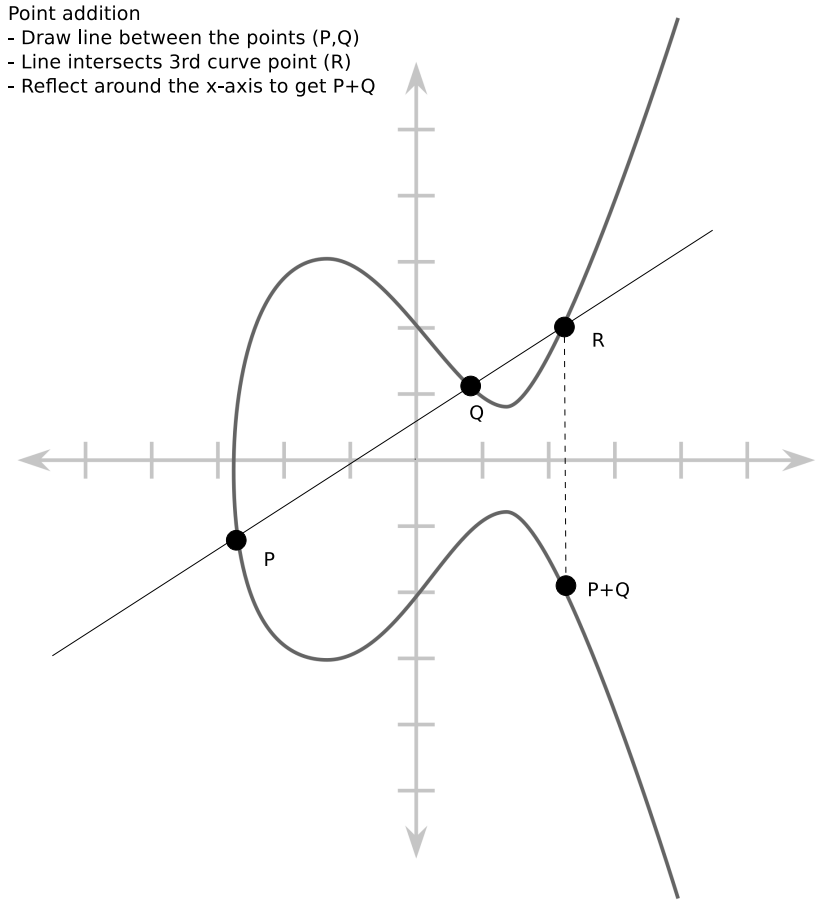

Figure 2: Point addition on elliptic curves over the real numbers

- Associativity: $P + (Q + R) = (P + Q) + R$
- Identity: the "zero" is the point at infinity ∞

Scalar multiplication is defined as a set of point additions to itself.

$$kP = P + P + P + ... + P(\text{k times}) \text{ for } k \neq 0$$

It follows that $nP = \infty$ if n is the order of the curve and $(n + 1)P = P$.

It is essential to provide special attention to the point at infinity ∞ when implementing any ECC-based applications, as obtaining this value in the middle of a cryptographic operation can bring serious security issues. This is particularly relevant because some of the issues discussed in Section 5.4 found during our tests occurred when ∞ appeared in an intermediate stage of scalar multiplications.

2.1.5 ECDHE

The Elliptic Curve Diffie-Hellman (ECDH) scheme [39, 56-58] was designed as an alternative for the DH algorithm over finite fields, but porting the parameters into

the ECC world. The principle is the same, but the underlying group is the set of points on an elliptic curve, such as P-256 [40]. After agreeing on the elliptic curve domain parameters to use, such as the specific curve to be used, a base point G, and a prime n of the order of G, the process is as follows:

1. Each party generates an elliptic curve key pair (d, Q), where d is a random integer in the interval $[1, n-1]$ acting as a private key, and $Q = d * G$ acts as the public key. Alice and Bob generate their pairs (d_A, Q_A) and (d_B, Q_B) respectively
2. Alice and Bob exchange their public keys Q_A and Q_B
3. Alice computes $S_A = d_A * Q_B$ and Bob computes $S_B = d_B * Q_A$
4. Just like before, since $S_A = d_A * Q_B = d_A * d_B * G = d_B * Q_A = S_B$, then S_A and S_B are equal. The x-coordinate of S_A and S_B is used as the shared secret

It is important to mention that public key validation is required by both parties when establishing session keys [41]. For DH, each party should verify that the public key they received belongs to an approved safe-prime group, while for ECDH, the key must correspond to the ECC group approved for key-establishment schemes. In particular, the parties need to make sure that their peer keys belong to the specific groups negotiated in the handshake. This is important in order to prevent, e.g., small subgroup attacks through malicious insertion of invalid keys or coding errors [39], and a number of validation methods have been standardized for this purpose, including the use of key validation primitives or receiving validation assurance from a trusted third party. One of the vulnerabilities found in the target system and discussed in Section 5.3.3 exemplifies the impact of missing these checks.

TLS 1.3 and earlier versions commonly use Ephemeral Diffie-Hellman (DHE) and Ephemeral Diffie-Hellman over Elliptic Curves (ECDHE) so each of the key pairs is short-lived, i.e., they expire after a short time. This ensures that the key pairs are frequently rotated so that the shared secrets are also different, enabling Forward Secrecy, i.e., past communications are kept secure since the attacker cannot decrypt the data without the ephemeral private keys. TLS versions below 1.3 also supported DHE and ECDHE but still allowed the non-ephemeral algorithms to be used.

Both ECDHE and DHE are supported by TLS 1.3 for key exchange, however, DHE is rarely used in practice due to it being much slower than ECDHE. The term (EC)DHE (with parentheses around "EC") denotes that either the finite fields or the ECC version can be used. TLS 1.3 performs the key exchange algorithm in one of three forms: (EC)DHE, PSK-only, or a combination of both PSK and (EC)DHE.

2.1.6 ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) [42] is the ECC version of the Digital Signature Algorithm (DSA), and, just like for (EC)DHE, they both rely on the DLP. Even though both ECDSA and DSA are included in the Digital Signature Standard (DSS) by NIST [43], TLS 1.3 includes ECDSA only. Like any

other cryptographic signature scheme, ECDSA is comprised of three algorithms: a key generation algorithm, a signing algorithm, and a verification algorithm [23, pp. 439-484]. Signature schemes define both endpoints of the communication as the *signer* (the party who creates a signature) and the *verifier* (the party who verifies the signature). Roughly, these algorithms work as follows:

Key generation. The original specification [42] states that both parties agree on the elliptic curve to be used, a generator point G on the curve, and the order n of subgroups for elliptic curve points that is also the one that defines the private key lengths. TLS 1.3, however, allows only standardized pre-defined curves to be used, such as those defined in [40]; each with their own parameters, generator and order. The parties then need to agree only on which of the allowed curves to use. This helps improve interoperability between different systems and reduce risks, since arbitrary curves can have weaknesses [38].

The *signer* chooses a random integer between 0 and $n-1$ as a private key d and calculates the value for the corresponding public key as

$$P = d * G$$

The resulting public key is a point on the curve, consisting of two coordinates $\{x, y\}$. As in any public key cryptography algorithm, the public key can be shared while the private key must be kept secret.

Signing. The *signer* calculates

$$h = H(M)$$

Where M is the message that the *signer* intends to sign and H is a cryptographic hash function ([42] originally stated that SHA-1 should be used, while [43] recommends to use any hash approved in [22]). The *signer* also generates a secure random number k between 1 and $n-1$ and calculates

$$R = G * k$$

Then, it assigns the x coordinate of R to r. Finally, the *signer* obtains s as

$$s = k^{-1}(h + r * d) \mod n$$

Where k^{-1} is the modular inverse of k such that $k * k^{-1} = 1 \mod n$. If at any point either r or s are equal to 0, the process needs to be restarted with a new k, however, this happens with negligible probability. The signature consists of both numbers $\{r, s\}$, which can be sent to the *verifier*.

Verification. The *verifier* can verify that the signature $\{r, s\}$ was created by the *signer* on the message M by performing the following process: First, the *verifier* generates h in the same way as done by the *signer*. Then, it gets the value s^{-1} as the modular inverse of s such that $s * s^{-1} = 1 \mod n$. Next, the *verifier* computes

$$u_1 = h * s^{-1} \mod n$$
$$u_2 = r * s^{-1} \mod n$$

Which is enough to obtain R', as

$$R' = u_1 * G + u_2 * P$$

Where P is the public key of the *signer*. Now, the *verifier* selects r' to be the x coordinate of R' and checks if $r' == r$, in which case the signature is correct.

To prove that the verification holds, we take

$$\begin{aligned} s^{-1} =& (k^{-1}(h + r * d) \mod n)^{-1} \\ =& k(h + r * d)^{-1} \mod n \end{aligned}$$

Then, we have (all operations happening modulo n but omitted for brevity):

$$\begin{aligned} R' &= u_1 * G + u_2 * P \\ &= (h * s^{-1}) * G + (r * s^{-1}) * P \\ &= (h * s^{-1}) * G + (r * s^{-1}) * d * G \\ &= s^{-1} * G * (h + r * d) \\ &= k * (h + r * d)^{-1} * G * (h + r * d) \\ &= k * G = R \end{aligned}$$

Which indicates that r' and r are equal. ECDSA signatures are commonly encoded in ASN.1/DER encoding (which will be discussed in more detail in Section 2.1.10) when being sent over a network.

2.1.7 ECC performance concerns

Given that ECC-based protocols rely on the elliptic curve DLP, they also rely on scalar multiplication arithmetic for setting security parameters and thus, performing these operations efficiently is a main concern for their design, especially when targeting embedded systems with low resources. As explained in Section 2.1.4, scalar multiplication, (i.e., the computation of $kP = P + P + ... + P$, k times, for an integer k and a point $P \in E(F_q)$ for a large q) is computed as a potentially large amount of point additions, which in turn consist of several field multiplications, inversions and squarings. Luckily, there are methods for optimizing the formulae used when computing ECC values.

The discussion of the following optimization methods is important to understand some of the vulnerabilities found in the target application, specially those explained in Section 5.4. These vulnerabilities are caused by implementation errors in libraries that use these optimized formulae, and can be abused to compromise the security of the channel with varying impact levels.

Using the notation from Rivain [44], we define

- **I** = cost of field inversion (e.g., $\frac{1}{x}$)
- **M** = cost of field multiplication (e.g., xy)
- **S** = cost of field squaring (e.g., x^2)
- **A** = cost of field addition, subtraction or doubling (e.g., $x + y$, $x - y$, $x + x$)

It is also assumed that the following approximations apply:

- $I \approx 100M$
- $S \approx 0.8M$
- A is negligible

These approximations were assumed by Rivain based on existing software implementations [44]. It can be concluded that the cost of field inversion **I** is the highest one and if it can be somehow reduced, then the overall cost of the scalar multiplication would decrease as well.

ECC points as have been discussed so far are represented in so-called *affine coordinates*, where a point P consists of two coordinates (x, y). However, there are other ways to represent ECC points that allow for arithmetic optimizations. *Projective coordinates* make use of a third coordinate Z, so that a point $P = (x, y)$ can be represented as $P = (X, Y, Z)$ where $x = \frac{X}{Z^c}$ and $y = \frac{Y}{Z^d}$ for some integers c and d. It can be noted that a single point P can have as many projective representations as there are different Z.

When the coordinates use $c = 2$ and $d = 3$, the point is said to be represented in *Jacobian coordinates*, which is the most widely used configuration for projective coordinates. These coordinates enable some speed improvements over point addition and doubling. Given a point $P = (X_1, Y_1, Z_1)$, the point doubling $P+P = (X_3, Y_3, Z_3)$ in Jacobian coordinates is defined by

$$\begin{aligned} X_3 &= B^2 - 2A \\ Y_3 &= B(A - X_3) \\ Z_3 &= Y_1 Z_1 \end{aligned}$$

with

$$\begin{aligned} A &= X_1 Y_1^2 \\ B &= \frac{1}{2}(3X_1^2 + aZ_1^4) \end{aligned}$$

where a comes from the short Weierstrass equation discussed earlier. This shows that Jacobian doubling does not require field inversions and it can be computed with a cost of $4M + 6S + 8A$. Furthermore, if $a = -3$, the formula for B changes to

$$B = \frac{1}{2}(3X_1^2 + -3Z_1^4) = \frac{3}{2}(X_1 + Z_1^2)(X_1 - Z_1^2) \quad (1)$$

which allows for the reduction of the Jacobian doubling cost to $4M + 4S + 9A$. When $a \neq -3$, it is still possible to reduce the cost if several consecutive point doublings are computed by keeping track of the value for aZ_1^4 at each addition and using it to compute the next as discussed in [44], resulting in a cost of $4M + 4S + 8A$.

For the Jacobian addition of $P + Q$ with $P \neq Q$ and $Q = (X_2, Y_2, Z_2)$, the formulae are defined as

$$\begin{aligned} X_3 &= F^2 - E^3 - 2BE^2 \\ Y_3 &= F(BE^2 - X_3) - DE^3 \\ Z_3 &= Z_1 Z_2 E \end{aligned}$$

with

$$\begin{aligned} A &= X_1 Z_2^2 \\ B &= X_2 Z_1^2 \\ C &= Y_1 Z_2^3 \\ D &= Y_2 Z_1^3 \\ E &= A - B \\ F &= C - D \end{aligned}$$

This addition can be computed with a cost of $12M + 4S + 7A$. If a mix of Jacobian and affine coordinates is used where Q is represented in affine coordinates making $Z_2 = 1$, the computation cost can be further reduced to $8M + 3S + 7A$.

The naive method of scalar multiplication for kP needs at least one point doubling and $k - 2$ point additions. Some of the methods that were developed to optimize this computation are the following:

Double-and-add. This algorithm, also known as the *binary scalar multiplication* algorithm [45], is based on the observation that

$$kP = \begin{cases} 2[k/2]P & ;\text{if } k \text{ is even} \\ 2[k/2]P + P & ;\text{if } k \text{ is odd} \end{cases}$$

For example, $26P = 2{*}13P, 13P = 2{*}6P + P, 6P = 2{*}3P$ and so on. This leads to the so-called *left-to-right* binary algorithm [44]. Similarly, there exists a *right-to-left* version governed by the equation $kP = \sum_i k_i[2^i]P$, where each k_i corresponds to a bit in the binary representation of k, i.e., $k_i \in 0, 1$ and $i < m - 1$ with m being the bit length of k. For example, $26P = (11010)_2 P = 16P + 8P + 2P$.

The pseudocode for the left-to-right *double-and-add* algorithm is as follows:

```
point double_and_add(point P, scalar s) {
  point R[0] = P;
  int m = getNumBits(s);
  int i, b;
  for (i = m-1; i >= 0; i--) {
    b = getBit(s, i);
    R[0] = 2*R[0];
    if (b == 1)
      R[0] = R[0] + P;
  }
  return R[0];
}
```

Computing a scalar multiplication with this algorithm requires on average m doublings and $m/2$ additions. Calculating, e.g., $26P$ requires 4 point doublings and 3 point additions, while using the naive method requires 1 point doubling and 24 point additions. A more detailed example is shown in Figure 3.

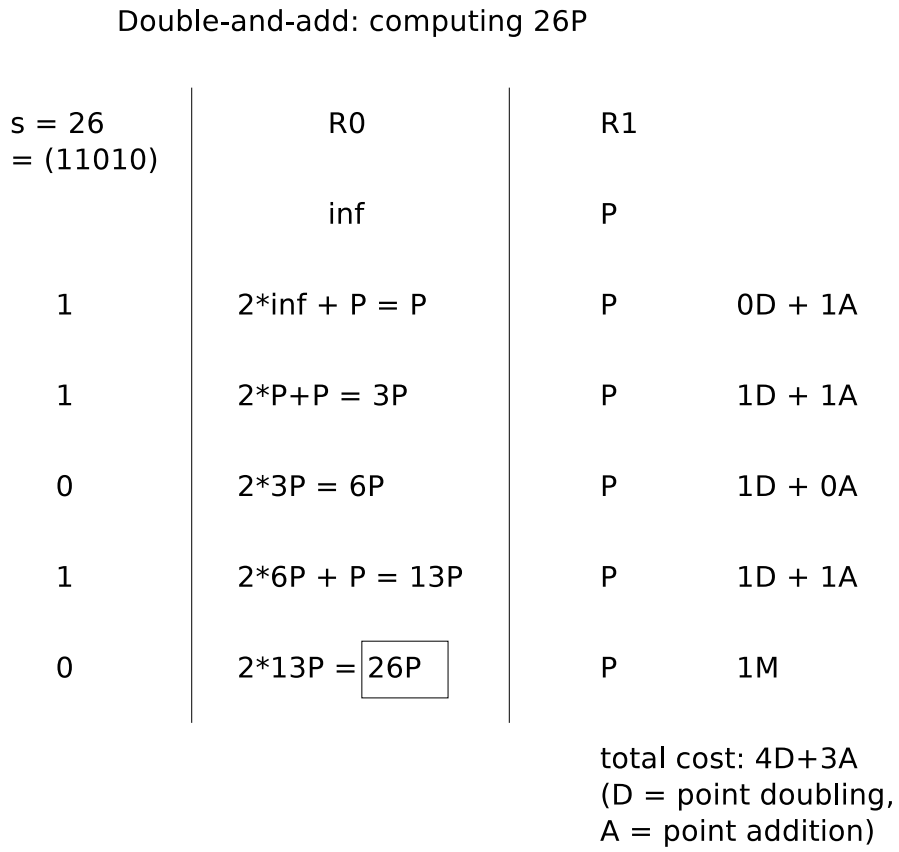

Figure 3: Double-and-add example

The double-and-add algorithm makes use of two point registers: R_0, which is used as an accumulator, and R_1, used as storage for the value of point P.

Double-and-always-add. The *double-and-add* algorithm is efficient, but also vulnerable to *side-channel attacks* [46, 238-250]. These attacks consider the possibility that the attacker might be able to obtain leaked information while the protocol is being executed, e.g., power consumption or electromagnetic radiation emissions in smart cards. Since point addition and point doubling computations yield different patterns in devices, a skillful attacker can potentially recover a secret scalar by tracing the information leaks to infer when the bits in the scalar are either one or zero.

A proposed solution for this problem was to integrate a "dummy" operation in the algorithm when a scalar bit is zero. The resulting algorithm was called the *double-and-always-add* algorithm [44], which does an addition and a doubling at each iteration, rendering information leakages useless. The pseudocode is shown below.

```
point double_and_always_add(point P, scalar s) {
  point R[2];
  R[0] = P; R[1] = P;
  int m = getNumBits(s);
  int i, b;
  for (i = m-1; i >= 0; i--) {
    b = getBit(s, i);
    R[0] = 2*R[0];
    R[1-b] = R[1-b] + P; // Dummy operation if b=0
  }
  return R[0];
}
```

This algorithm, however, is not very efficient as it results in a computation cost of $12M + 7S + 19A$ per iteration.

Montgomery Ladder. The *Montgomery ladder* [44] is a variant of the double-and-add algorithm with no dummy operations, i.e., every operation affects the result, which makes side-channel attacks harder. At every iteration there is a point addition and a point doubling, and the registers always satisfy the relation $R_1 - R_0 = P$. The pseudocode is as follows:

```
point montgomery_ladder(point P, scalar s) {
  point R[2];
  R[0] = P; R[1] = 2P;
  int m = getNumBits(s);
  int i, b;
  for (i = m-2; i >= 0; i--) {
    b = getBit(s, i);
    R[1-b] = R[1-b] + R[b];
    R[b] = 2*R[b]; // All operations are effective
  }
  return R0;
}
```

An example of the algorithm is shown in Figure 4.

Montgomery ladder: computing 26P

s = 26 = (11010)	R0	R1
	P	2P
1	P + 2P = 3P	2*2P = 4P
0	2*3P = 6P	3P + 4P = 7P
1	6P + 7P = 13P	2*7P = 14P
0	2*13P = 26P	

Figure 4: Montgomery ladder example

2.1.8 ECC Co-Z Addition

Another optimized set of formulae specifically for the Jacobian addition, is the *Co-Z Addition* formulae introduced by Méloni [47]. This method, referred to as ZADD, considers the case when the Z-coordinate is the same for both P and Q. In this case, the definition for the sum $P + Q = (X_3, Y_3, Z_3)$ where $P = (X_1, Y_1, Z)$ and $Q = (X_2.Y_2, Z)$ is

$$
\begin{aligned}
X_3 &= D - (B + C) \\
Y_3 &= (Y_2 - Y_1)(B - X_3) - E \\
Z_3 &= Z(X_2 - X_1)
\end{aligned}
$$

with

$$
\begin{aligned}
A &= (X_2 - X_1)^2 \\
B &= X_1 A \\
C &= X_2 A \\
D &= (Y_2 - Y_1)^2 \\
E &= Y_1(C - B)
\end{aligned}
$$

This formulae are more efficient than the ones presented previously, with a computation cost of $5M + 2S + 7A$. Additionally, the Co-Z addition has the benefit of allowing an update "for free" of the point P to a value that shares the Z-coordinate with the result. This is possible because of the formulae used for B and E:

$$B = X_1 A = X_1(X_2 - X_1)^2 = X_1' Z_3^2$$
$$E = Y_1(C - B) = Y_1 A(X_2 - X_1) = Y_1(X_2 - X_1)^3 = Y_1' Z_3^3$$

where $(X_1', Y_1') = (X_1 Z^{-2}, Y_1 Z^{-3})$, so P' can be represented as $P' = (B, E, Z_3)$. Since now P' and $P + Q$ share the same z-coordinate, this allows for subsequent additions between these points still using the Co-Z addition. This Co-Z addition operation with point update is denoted as ZADDU, then, $ZADDU(P, Q) = (P', P + Q)$ where the Z-coordinate of P' is the same than the one for $P + Q$.

Furthermore, Goundar et al. [45] introduced yet another variant called the *conjugate Co-Z addition*, denoted ZADDC. This variant calculates the result for $P + Q$ simultaneously as for $P - Q$, where both results share the same Z-coordinate, thus $ZADDC(P, Q) = (P + Q, P - Q)$. This is achieved by observing that if $-Q = (X_2, -Y_2, Z)$, then $P - Q$ can be denoted as (X_3', Y_3', Z_3) where

$$X_3' = (Y_1 + Y_2)^2 - (B + C)$$
$$Y_3' = (Y_1 + Y_2)(B - X_3') - E$$

Obtaining these values from $P + Q$ in ZADD involves an additional cost of $1M + 1S + 3A$, giving a total computation cost of $6M + 3S + 11A$ for ZADDC. Now, ZADDC can be used in combination with the Montgomery ladder discussed in the previous section by rewriting the steps in the loop as

$$R[1 - b] \leftarrow R[1 - b] + R[b]$$
$$R[b] \leftarrow 2 * R[b] = R[b] + R[1 - b] + R[b] - R[1 - b]$$

Therefore, having a temporary point $T = R[b] - R[1 - b]$,

$$(R[1 - b], T) \leftarrow ZADDC(R[b], R[1 - b])$$
$$R[b] \leftarrow R[1 - b] + T$$

Given the nature of ZADDC, $R[1 - b]$ and T share the same z-coordinate, so ZADD can be used to obtain $ZADDU(R[1 - b], T) \leftarrow (R[b], R[1 - b])$ where the results also share the z-coordinate. It is evident that the process can be iterated, reusing the Co-Z addition variants in every iteration. This combination of the Montgomery ladder and Co-Z additions has a final cost of $9M + 7S + 27A$ after some standard reductions.

Venelli and Dassance [48] propose a set of algorithms using the Co-Z addition that perform iterative operations on the (X, Y) coordinates of two co-z points and then compute Z in a final step. These algorithms were later called the *(X,Y)-only co-Z addition* variants, which consist mainly of the so-called $XYZC - ADD$ algorithm as the (X,Y)-only version of ZADDU, and the $XYZC - ADDC$ algorithm for the

ZADDC counterpart. The Z-coordinate can be recovered right before the last addition by using the formula

$$Z = X_b y_P (x_P Y_b (X_1 - X_0))$$

where (X_b, Y_b) are the coordinates in register $R[b]$ and (x_P, y_P) are the affine coordinates of the original point P. The formula is based on the Montgomery ladder invariant $R_1 - R_0 = P$.

2.1.9 ECDSA verification optimizations

As discussed in Section 2.1.6, an ECDSA signature consists of two integers $\{r, s\}$ and the verification of the signature over a given message is done by computing $R' = u_1 * G + u_2 * P$, with

$$\begin{aligned} u_1 &= h * s^{-1} \mod n \\ u_2 &= r * s^{-1} \mod n \end{aligned}$$

where G is a generator point in the selected curve, P is the public key of the signer, h is the cryptographic hash of the message, and s^{-1} is the modular inverse of s. It is evident that u_1 and u_2 are integers while G and P are points in the curve. A normal computation for R' requires two scalar multiplications and one point addition.

There is an optimized method that helps speed up this operation known as *Shamir's trick*, which reduces the computation cost to almost the same as one scalar multiplication [46, pp. 109-113]. This method requires that $P + G$ is computed in advance and then runs through the bits of u_1 and u_2 and performs an iterative sum of either G, P, or $P + G$ based on which bits are set, doing so a simultaneous multiplication of the points. The pseudocode of the algorithm is as follows:

```
point shamirs_trick(point G, point P, scalar u1, scalar u2) {
  point R[3], S;
  R[0] = 0; R[1] = G;
  R[2] = P; R[3] = P+G;
  int m1 = getNumBits(u1);
  int m2 = getNumBits(u2);
  int m = max(m1,m2);
  int i, b;
  S = P;
  for (i = m-2; i >= 0; i--) {
    b = (getBit(u1, i) << 1) | getBit(u2, i);
    S = S + R[b];
  }
  return S;
}
```

Given that this algorithm results in computation cost savings most of the time, it is often included in optimized software implementations of ECC operations.

Type	Tag number (decimal)	Tag number (hexadecimal)
INTEGER	2	02
BIT STRING	3	03
OCTET STRING	4	04
NULL	5	05
OBJECT IDENTIFIER	6	06
SEQUENCE and SEQUENCE OF	16	10
SET and SET OF	17	11
PrintableString	19	13
T61String	20	14
IA5String	22	16
UTCTime	23	17

Table 1: Some ASN.1 types and their tags. Adapted from [49]

2.1.10 ASN.1

The Abstract Syntax Notation One (ASN.1) is a language used to define abstract objects for the Open Systems Interconnection (OSI) architecture [49]. This notation is particularly useful when defining data that is sent over a network. ASN.1 is defined in the X.680 standard of ITU-T [50]. It uses a combination of types and values to describe the objects and allows to name types (using the assignment operator ::=) which can be re-used for defining additional types or objects. Most ASN.1 encodings consist of a class and a tag number (generally represented in hexadecimal characters). Some of the most used types are shown in Table 1

ASN.1 classifies its available types in four classes: simple types, structured types, tagged types, and other types. The types that are mostly used to represent objects in TLS-related cryptographic algorithms and protocols (and thus, the ones that are most relevant for this thesis) are the ones listed in Table 1, and mainly: SEQUENCE (of the structured type class) and INTEGER, BIT STRING, and OCTET STRING (of the simple type class). The ASN.1 description that will be used for abstract objects in this thesis follows the format:

```
SequenceName ::= SEQUENCE {
    valueName1  valueType1,
    ...         ...,
    valueNameN  ValueTypeN
}
```

The ASN.1 notation for some of the abstract objects that were explained in previous sections are shown next.

ECDSA Signature [42]

```
ECDSA-Sig-Value ::= SEQUENCE {
```

```
    r       INTEGER,
    s       INTEGER
}
```

SubjectPublicKeyInfo [51]

```
SubjectPublicKeyInfo  ::=  SEQUENCE  {
    algorithm               AlgorithmIdentifier,
    subjectPublicKey        BIT STRING
}

AlgorithmIdentifier   ::=  SEQUENCE  {
    algorithm               OBJECT IDENTIFIER,
    parameters              ANY DEFINED BY algorithm OPTIONAL
}
```

The optional parameters vary according to the algorithm.

X.509 Certificate [52]

```
Certificate  ::=  SEQUENCE  {
    tbsCertificate       TBSCertificate,
    signatureAlgorithm   AlgorithmIdentifier,
    signatureValue       BIT STRING
}

TBSCertificate  ::=  SEQUENCE  {
    version         [0]  EXPLICIT Version DEFAULT v1,
    serialNumber         CertificateSerialNumber,
    signature            AlgorithmIdentifier,
    issuer               Name,
    validity             Validity,
    subject              Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                         -- If present, version MUST be v2 or v3
    subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                         -- If present, version MUST be v2 or v3
    extensions      [3]  EXPLICIT Extensions OPTIONAL
                         -- If present, version MUST be v3
}
```

where each additional component (with the exception of `AlgorithmIdentifier` and `SubjectPublicKeyInfo`, which are defined above) is defined as

```
Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }

CertificateSerialNumber  ::=  INTEGER

Validity ::= SEQUENCE {
     notBefore      Time,
     notAfter       Time
 }

Time ::= CHOICE {
     utcTime        UTCTime,
     generalTime    GeneralizedTime
 }

UniqueIdentifier  ::=  BIT STRING

Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE  {
     extnID      OBJECT IDENTIFIER,
     critical    BOOLEAN DEFAULT FALSE,
     extnValue   OCTET STRING
                 -- contains the DER encoding of an ASN.1 value
                 -- corresponding to the extension type identified
                 -- by extnID
 }
```

ASN.1 has two main sets of rules for representing abstract objects as bit strings for their use in computer networks: the Basic Encoding Rules (BER) and, a subset of BER, the Distinguished Encoding Rules (DER), both defined in the X.690 standard [53]. These rules define how to encode ASN.1 types into sequences of 8-bit octets. Both BER and DER use the *Tag-Length-Value* (TLV) approach, where for each encoded value the first field corresponds to the tag or type, which tells the decoder or parser how to interpret the data; the second field corresponds to the length of the value, which the decoder can use to know how many bytes to read; and the last field corresponds to the value itself.

In BER, three methods are used for encoding of ASN.1 objects: primitive with definite-length, constructed with definite-length, and constructed with indefinite-length. The first octet is used to identify the class of an object. The class is defined using the first two bits of the octet, Bit 8 and Bit 7. The available classes are shown in Table 2.

Bit 6 in the identifier octet is set to "0" when the encoding is primitive and "1" when it is constructed. Bits 5 to 1 are used to define the tag number, from the ones shown in Table 1. The tag number can have one of two forms: low tag number, for tag numbers below 30, and high tag number, for numbers equal to or greater than

Class	**Bit 8**	**Bit 7**
universal	0	0
application	0	1
context-specific	1	0
private	1	1

Table 2: Classes in identifier octets [50]

Identifier octet							
8	7	6	5	4	3	2	1
Class		P/C	Tag number				

Table 3: Bits in BER encoding identifier octet [50]

31. When the high tag number form is used, all Bits 5 to 1 are set to "1" and the following octets indicate the tag number, where all octets except for the last have bit 8 set to "1". Table 3 summarizes the bits positions and uses in the identifier octet for BER.

In an ECDSA signature, for example, the BER encoding identifier octet is generally set to "00110000" which translates to the hex value 0x30. This determines that the object has a universal class (value "00" for bits 8 and 7), is constructed (value "1" for bit 6), and is of type SEQUENCE (value "10000" for bits 5 to 1 which equals 0x10 in hexadecimal and 16 in decimal).

Following the TLV format, the length octets appear immediately after the identifier or tag octets. Length octects can also be in one of two forms: short form, for values lower than 127; and long form, for values equal to or greater than 128. For the long form, bit 8 is set to "1", while the bits 7 to 1 indicate how many additional length octets will be used, which are the ones that give information on the actual size of the data. When the length of the data is not known beforehand, e.g., when using the constructed with indefinite-length method, the length octet is just one and is set to $0x80$. This, however, requires that the data ends with an *end-of-content* field comprised of two octets set to 0x00.

After the length octets come the bytes of the actual value. As mentioned before, a parser would know how many bytes to read as the value based on what was specified in the length octets. In particular, INTEGER values are written as a two's complement binary number, so they use the highest bit as a "sign indicator", i.e., if the highest or leftmost bit is set to "1", the value should be interpreted as a negative number. For this reason, if a positive integer has a highest bit set to "1", it should be preceded by a $0x00$ byte to avoid incorrect interpretations [53, p. 7]. Additionally, INTEGER values should be encoded with the smallest number of octets possible, so no unnecessary zero padding bytes are allowed.

As mentioned earlier, the DER encoding is a subset of BER, and therefore, all DER-encoded objects are also valid BER-encoded objects. DER follows the same rules as BER but has the following additional restrictions [53]:

- The definite method for length octets must be used, having as minimum amount

of octets as possible (indefinite length is not allowed).

- String types of BIT STRING, OCTET STRING and restricted character types must use the primitive form.
- The values composing a SET should be ordered according to their tags.

Further extending the ECDSA signature example, this is a valid ECDSA signature encoded in DER in hexadecimal representation:

```
3044 0220 5031 3f3f 3f3f 3f3f 1b3f 0f1d 3f3f 673f 3f08 483f
4578 3f3f 3607 3f3f 3f3b 5647 0220 493f 3f3f 3f2a 224f 533f
0170 143f 193f 5447 6554 7a3f 7c3f 3f20 3f3f 583f 3f3f
```

The signature begins with a 0x30 byte which denotes a universal constructed SEQUENCE. The next byte, 0x44 or 68 in decimal, states the length for the rest of the bytes in the SEQUENCE. The next byte in the stream is a 0x02, indicating that the first element in the SEQUENCE is an INTEGER, followed by a 0x20 or decimal 32 stating that the following 32 bytes correspond to the value of the first INTEGER in the SEQUENCE (starting with 0x50). After these 32 bytes, there is another 0x02 and 0x20 indicating that another 32 byte INTEGER follows. Looking at the ASN.1 structure for ECDSA signatures shown earlier, it can be concluded that the values of the signature are

```
r = 0x50313f3f3f3f3f3f1b3f0f1d3f3f673f3f08483f45783f3f36073f3f3f3b5647
s = 0x493f3f3f3f2a224f533f0170143f193f544765547a3f7c3f3f203f3f583f3f3f
```

2.2 The SSL/TLS protocol

SSL and TLS are both cryptographic client-server protocols that were designed to protect data communications against unauthorized reading, modification, and forgery. The first version of the protocol to be released to the public was launched in 1994 by Netscape Communications [54, pp. 3-4]. This version was SSL version 2 (SSLv2), as the first version was never actually released. At the time, SSLv2 was intended to protect traffic for web, mail and news in the Internet, and Netscape meant to use it for other protocols beside HTTP. However, the protocol was developed without enough security testing and validation and was shown to have several security issues.

In order to cope with the discovered vulnerabilities and provide fixes, Netscape created SSL version 3 (SSLv3) [55] just one year later. This version was not only an improvement over SSLv2, but it also had new mechanisms and design goals. Particularly, SSLv3 includes the capability of negotiating multiple cryptographic algorithms in a secure manner [56, pp. 44-45]. Netscape decided to hire security consultants, including well-known cryptography experts such as Taher Elgamal [57, p. 15], to help in the development process, which resulted in a more secure protocol that was very different to the previous SSLv2 version.

During this process, other vendors were implementing the protocol to use with their systems, while some were adding extra features. In particular, Microsoft wanted to add more functionalities to the protocol that they considered critical [56, pp. 49-52]. This

led the Internet Engineering Task Force (IETF) to organize a working group and a set of meetings that resulted in the creation of the TLS protocol, in an attempt to combine the work from both Microsoft and Netscape and to generate a standardized version of the protocol.

The TLS 1.0 version [58] was then released in 1999 without many changes in comparison with the previous version and acting more as a cleanup and rename of the protocol. In 2006, TLS 1.1 [59] got released with new security fixes and a new feature called the *TLS Extensions* [60] [54, pp. 3-4], which can be used to provide additional functionalities and mechanisms negotiated during the protocol handshake. Later, in 2008, TLS 1.2 [61] was released bringing once more extra security fixes, removing deprecated protocols, and supporting authenticated encryption.

Finally, TLS 1.3 [2] was released 10 years later in 2018 and it is the latest version at the time of writing this thesis. This version contains several differences compared to the previous ones, including substitution of legacy cipher suites (e.g., suites with RSA key transport, static DH, and MAC-then-encrypt suites such as AES_CBC) with Authenticated Encryption with Associated Data (AEAD) algorithms, addition of forward secrecy for all key exchange mechanisms, confidentiality for all handshake messages succeeding the `ServerHello` message, use of the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) as underlying primitive for Key Derivation Functions (KDF), and deprecation of the version negotiation mechanism introduced in TLS 1.2 which was a frequent source of software bugs.

2.2.1 The TLS Handshake

In the TLS handshake sub-protocol, the parameters for the connection to be established are negotiated between the client and server, and the application traffic protection keys are derived. The TLS handshake is meant to be tamper-resistant, i.e., an attacker should not be able to force the communication parties to negotiate any parameters or cryptographic modes that are different to what they would set if the attacker was not present [2].

TLS 1.3 defines a set of handshake flow variants to be used depending on the channel requirements, e.g., for server-only authentication, for both server and client authentication, for renegotiation in case of insufficient data, for Pre-Shared Key (PSK) authentication, etc. Figure 5 shows the basic full TLS handshake with certificate-based server authentication. This handshake is initiated by the client and is commonly used when there is no shared key data between both parties, such as when both endpoints establish a secure connection for the first time.

The basic full TLS handshake has 3 *flights* (or sets of consecutive messages) sent by each party. In the first flight, the client notifies the server that it wants to establish a secure connection with a `ClientHello` message that includes the following fields, among others: 1) a random nonce, 2) the list of supported cipher suites specifying the AEAD and HKDF algorithms, 3) a list of supported (EC)DHE groups and (EC)DHE key share for the groups (in a `key_share` extension), and 4) a list of supported signature algorithms.

In the second flight, the server sends the `ServerHello` message where it specifies the selected connection parameters and sends its own (EC)DHE `key_share`. At this point, both the client and server have enough information to derive their (EC)DHE handshake keys. Still as part of the second flight, the server sends a `Certificate` message containing its own certificate, a `CertificateVerify` message containing a signature over the transcript of the handshake messages exchanged so far computed with the private key that corresponds

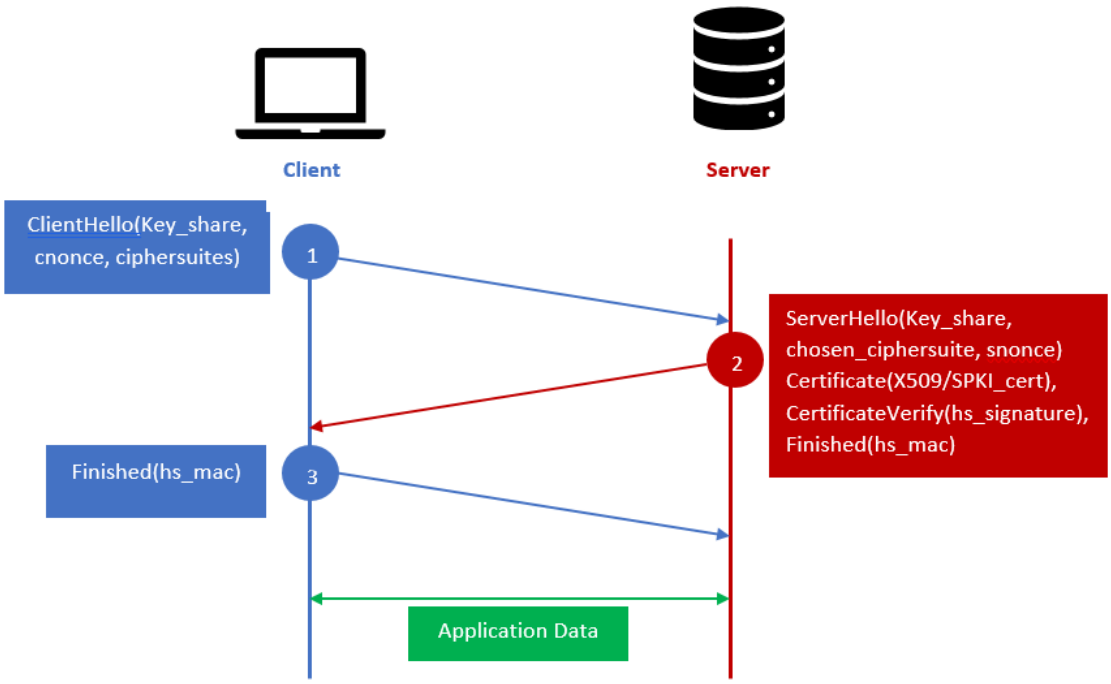

Figure 5: Basic full TLS handshake with server authentication

to the certificate, and a `Finished` message with a Message Authentication Code (MAC) of the handshake, having all three of these messages now encrypted with the handshake key.

For the final step, the client receives the server certificate and can verify that it is valid, thus authenticating the server. In the third flight, the client sends its own `Finished` message, confirming the validity of the exchanged parameters and finishing the handshake. Once the handshake is done, both parties derive the keys required by the record protocol and can send protected application data.

As mentioned before, variants of the handshake flow can be used on different situations or problems that may be encountered during the communication. These are some of the variants that can occur during the TLS handshake:

Incorrect DHE share. If the `key_share` sent by the client is not supported by the server or is insufficient, the server can send a `HelloRetryRequest` back to the client before proceeding to phase two of the handshake. The client then needs to initiate a new handshake while sending new security parameters to try and match with the ones supported by the server. If both client and server cannot agree on a set of security parameters, the handshake must be aborted.

Client authentication. The server can request that the client is also authenticated. In this case, the server will send a `CertificateRequest` message during the second flight which contains the required parameters for the client certificate. The client then needs to send `Certificate` and `CertificateVerify` messages in a similar way as done by the server. The server can then verify the data and authenticate the client. Invalid certificates

or certification paths will result in the abortion of the handshake.

PSK authentication. The server and the client can have a symmetric key that was shared before the communication. Assuming the method used for sharing the key can be trusted, mutual proof of possession for the PSK is enough for authentication. However, the PSK can be used in combination with (EC)DHE for providing forward secrecy. The client specifies during the initial phase whether a PSK will be used and sends an identifier or list of identifiers of the PSK to be used. The server then selects a PSK and sends back a message stating the selection during flight two and the handshake continues as before. If both PSK and (EC)DHE are being used, the PSK-related messages are sent along with the (EC)DHE parameters using their respective TLS extensions. Using a PSK provides certain advantages, e.g., the certificates can be skipped completely; PSKs can also be used for session resumption.

Session resumption. PSKs can be established in an out of band channel, but they can also be generated and exchanged after a full handshake has been finished. These PSKs can also be used by the client for authentication, in which case the new connection is bound to the previous one and a full handshake does not need to be performed, so the previous session is "resumed". When using a PSK, the server can start sending encrypted data from the second flight of the handshake. The client should still send a `key_share` extension to allow the possibility of falling back to a full handshake in case the server declines the resumption. This also allows the server to send its own `key_share` data to provide forward secrecy.

0-RTT. The Zero Round-Trip Time (0-RTT) is a feature offered by TLS 1.3 that allows a client to send encrypted data from the first phase of the handshake when both client and server already have a PSK. This allows to save time in the communication, however, the encrypted data sent this way is not as secure as data sent after a proper handshake. The data sent in 0-RTT is encrypted solely with the PSK so it does not provide forward secrecy. Also, there is no replay protection for 0-RTT messages since this protection is added after the second flight when the server has sent a random nonce.

Attested TLS. The goal of TLS is to provide a secure channel for data communication between two parties. A secure channel, however, does not take into account endpoint integrity, i.e., it does not provide protection against attacks enabled by compromised endpoints, such as malicious data or code injections. The combination of a secure channel with endpoint integrity, also known as *trusted channel*, can be achieved with the use of *attestation*, which refers to the process of providing verifiable integrity guarantees on top of the secure channel with the use of trusted system components, such as a Hardware Security Module (HSM) or Trusted Platform Module (TPM). There are a number of proposals to integrate TLS with attestation, including the use of the Trusted Sockets Layer (TSL) protocol which adds steps for generation and verification of attested evidence exchanged during a TLS handshake [20].

2.2.2 The TLS Presentation Language

The TLS protocol uses a specific presentation syntax similar to ASN.1, discussed in Section 2.1.10, to represent the objects that both parties exchange during a session. This

presentation language defines the structure of the objects and types that are then translated into bytes and commonly represented with hexadecimal characters. This section shows the common types and structures used by TLS 1.3 as defined in the RFC [2].

Integers are defined as concatenations of bytes with a fixed length. These concatenations are constructed from the basic unit of a single unsigned byte or 8-bits *uint*8, and thus their bit length must be a multiple of 8. The maximum numeric type in TLS has a length of 8 bytes or 64 bits. All objects in bytes transmitted over the network are sent in big-endian order. These are the predefined numeric types:

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

The brackets [] are used to indicate a vector or an array of a single dimension. The number between the brackets indicates the number of bytes in the vector. This is used to denote new types that are composed of several bytes of another type. Vectors of variable length can be defined using the notation $< floor..ceiling >$ instead of the brackets, where the $floor$ value denotes the minimum amount of bytes in the vector and the *ceiling* value denotes the maximum amount of bytes. The syntax to denote a new type T' as a vector of fixed length n or variable length from n to m bytes of a type T is as follows:

```
T T'[n];        /* fixed-length vector */
T T'<n..m>;     /* variable-length vector */
```

The characters $/*$ and $*/$ are used to denote comments. When encoding for network transmission, variable-length vectors must be prepended with an extra field that indicates the actual length of the vector. The type *opaque* is used to represent objects composed of a single byte that contain data that is not interpreted. For example, the following syntax denotes a type called *mandatory* which is a vector of uninterpreted data that must be between 300 and 400 bytes in length:

```
opaque mandatory<300..400>;
```

The *enumerated* type defines a set of elements where each has an assigned value. These values can later be assigned or compared but only with other *enumerated* elements of the same type. The syntax for an *enumerated* type Te where each en represents an element and each vn represents its value is shown below, the symbols [[and]] denote optional data.

```
enum {
    e1(v1),
    e2(v2),
    ... ,
    en(vn)
    [[, (n)]]
} Te;
```

The maximum value for a set of elements in an *enumerated* defines the amount of bytes that the *enumerated* would occupy in the byte stream. It is also possible to define a value without an element name to force the byte length in the stream. Element names can be repeated and the numerical values can also describe ranges of numbers over which the same element name applies. The following example denotes an *enumerated* type `Color` with values for the elements `red`, `blue`, and `white`. These values can be accessed explicitly using a syntax, such as `Color.blue`, or implicitly using the element name `blue` if it is clearly specified what the target of the assignment is. Also, since the maximum value for the enumerated is 7, which fits in one single byte, the `Color` enumerated would occupy just one byte in the stream.

```
enum {
    red(3),
    blue(5),
    white(7)
} Color;
```

Similarly as in the C language, *structure* types can be defined as a construction of other primitive types. The syntax for a new structured type T composed of Tn types with respective fn fields is as follows:

```
struct {
  T1 f1;
  T2 f2;
  ...
  Tn fn;
} T;
```

Each *struct* element or field can be a vector as well and can be accessed with a syntax similar to that used in *enumerateds*, i.e., `T.f2`. Fields in a *struct* can be assigned a fixed value using the equal operator =. Structures can have a field called a *variant* which changes depending on a *selector* and the value it has. The *selector* is of type *enumerated* and defines the conditions for which the value of the *variant* changes and their respective types. The syntax is as follows:

```
struct {
  T1 f1;
  T2 f2;
  ....
  Tn fn;
  select (E) {
    case e1: Te1 [[fe1]];
    case e2: Te2 [[fe2]];
    ....
    case en: Ten [[fen]];
  };
} Tv;
```

2.2.3 TLS Structures

This subsection explains some of the most important structures defined in the TLS 1.3 RFC [2]. These structures are transmitted over the network when performing handshakes and exchanging data, and they need to be encoded in specific ways for this transmission. They are also later used in Section 4.2.3 when each of the messages in a handshake are fuzzed as part of the testing framework.

ClientHello

```
uint16 ProtocolVersion;
opaque Random[32];

uint8 CipherSuite[2];     /* Cryptographic suite selector */

struct {
  ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
  Random random;
  opaque legacy_session_id<0..32>;
  CipherSuite cipher_suites<2..2^16-2>;
  opaque legacy_compression_methods<1..2^8-1>;
  Extension extensions<8..2^16-1>;
} ClientHello;
```

This is the `ClientHello` message initially sent by the client to begin the handshake protocol. The `legacy_version` field is a 16 bit that previous protocol versions used to negotiate the version of TLS to use, however, due to frequent implementation problems, this field is no longer used for that purpose. TLS 1.3 fixes the `legacy_version` to 0x0303, which is the value assigned for TLS 1.2. The way to identify a TLS 1.3 `ClientHello` message is by looking for a `supported_versions` Extension with a value of 0x0304 as the highest version.

The `random` field is a 32 bytes long value generated by a secure PRNG, and is generally used for generating keying material. The `legacy_session_id` is a value between 0 and 32 bytes long that was used in previous versions of TLS for session resumption. TLS 1.3 no longer uses it for this purpose as this functionality is provided with PSKs, but it still can use this field for compatibility with older versions. The `cipher_suites` field contains the list of supported cipher suites that the client wishes to use. Each cipher suite takes a space of two bytes in the stream and the first two bytes of the field are reserved for specifying the actual length of the list, which gives information on the amount of cipher suites being sent.

The `legacy_compression_methods` is another field that was used in previous versions of TLS to indicate the compression methods to use, but for TLS 1.3 it must be set to zero in a single byte, while any other value should be rejected. The `extensions` field is used to request extended TLS functionality. The format of this field follows another structure format explained below.

ServerHello

```
struct {
  ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
```

```
  Random random;
  opaque legacy_session_id_echo<0..32>;
  CipherSuite cipher_suite;
  uint8 legacy_compression_method = 0;
  Extension extensions<6..2^16-1>;
} ServerHello;
```

The `ServerHello` is sent by the server in response to a `ClientHello` message. Just like the client counterpart, `ServerHello` contains a `legacy_version` field set to 0x0303 from TLS 1.2 and a `random` field with a PRNG-generated 32 bytes long value. The `legacy_session_id_echo` field contains the same session ID sent by the client; if the server sends a different value, the client must abort the handshake. The `cipher_suite` field is the cipher suite selected by the server from the list sent by the client; if the value sent by the server was not in the original list, the client must abort the handshake. Since the compression value is not used in TLS 1.3, the `legacy_compression_method` field sent by the server must be a single zero byte. The `extensions` field contains data used by extended functionality requests, however, this field must have only data relevant to the protocol negotiation and definition of cryptographic context, data for other extensions is sent in a separate `EncryptedExtensions` message. Additionally, the `extensions` field must contain a `supported_versions` extension that identifies the protocol in use as TLS version 1.3.

HelloRetryRequest The `HelloRetryRequest` uses the same structure as the `ServerHello` message and is sent when the `ClientHello` message does not provide enough data to continue the handshake. The fields in this message have the same meaning as in the `ServerHello` message. When receiving a `HelloRetryRequest` from the server, the client needs to resend the `ClientHello` message with new security parameters to try and fulfill the requirements that were not previously met so the handshake can proceed. The `HelloRetryRequest` message can only be sent once per handshake.

Extensions

```
struct {
  ExtensionType extension_type;
  opaque extension_data<0..2^16-1>;
} Extension;

enum {
  server_name(0),
  max_fragment_length(1),
  status_request(5),
  supported_groups(10),
  signature_algorithms(13),
  use_srtp(14),
  heartbeat(15),
  application_layer_protocol_negotiation(16),
  signed_certificate_timestamp(18),
  client_certificate_type(19),
```

```
  server_certificate_type(20),
  padding(21),
  pre_shared_key(41),
  early_data(42),
  supported_versions(43),
  cookie(44),
  psk_key_exchange_modes(45),
  certificate_authorities(47),
  oid_filters(48),
  post_handshake_auth(49),
  signature_algorithms_cert(50),
  key_share(51),
  (65535)
} ExtensionType;
```

The extensions sent in the `extensions` field of the initial handshake messages need to follow the `Extension` structured format shown above. This format has an `extension_type` field which indicates the type of the extension based on the `ExtensionType` enumerated, and an `extension_data` field which contains actual data used by the extension. The format of each extension data varies depending on their type and they will not be discussed here.

Certificate

```
enum {
  X509(0),
  RawPublicKey(2),
  (255)
} CertificateType;

struct {
  select (certificate_type) {
    case RawPublicKey:
      /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
      opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

    case X509:
      opaque cert_data<1..2^24-1>;
  };
  Extension extensions<0..2^16-1>;
} CertificateEntry;

struct {
  opaque certificate_request_context<0..2^8-1>;
  CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

The `Certificate` message is sent by the server so the client can verify its identity. This message can only be omitted if the authentication is done with PSKs. The server can also request that the client uses a certificate for authentication by sending a `CertificateRequest`

message, which a client must respond with its corresponding `Certificate` message; this is the only valid situation in which a client should send a `Certificate` message. The verifying party must check the validity of all certificates sent in the chain and their signatures. If an error is found, the handshake must be aborted.

The `certificate_request_context` field must be set to zero length unless the message is a response to a `CertificateRequest` message, in which case it must have the same value as the field with the same name in the request. The `certificate_list` contains the certificate chain, each as a `CertificateEntry` structure. A `CertificateEntry` structure can additionally have a set of extensions in the `extensions` field, following the format shown earlier. At the time of writing, the supported public key formats are X.509 [52] and ASN.1 SubjectPublicKeyInfo [51] (the ASN.1 structures for both of these objects were shown in Section 2.1.10).

CertificateVerify

```
enum {
  /* RSASSA-PKCS1-v1_5 algorithms */
  rsa_pkcs1_sha256(0x0401),
  rsa_pkcs1_sha384(0x0501),
  rsa_pkcs1_sha512(0x0601),

  /* ECDSA algorithms */
  ecdsa_secp256r1_sha256(0x0403),
  ecdsa_secp384r1_sha384(0x0503),
  ecdsa_secp521r1_sha512(0x0603),

  /* RSASSA-PSS algorithms with public key OID rsaEncryption */
  rsa_pss_rsae_sha256(0x0804),
  rsa_pss_rsae_sha384(0x0805),
  rsa_pss_rsae_sha512(0x0806),

  /* EdDSA algorithms */
  ed25519(0x0807),
  ed448(0x0808),

  /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
  rsa_pss_pss_sha256(0x0809),
  rsa_pss_pss_sha384(0x080a),
  rsa_pss_pss_sha512(0x080b),

  /* Legacy algorithms */
  rsa_pkcs1_sha1(0x0201),
  ecdsa_sha1(0x0203),

  /* Reserved Code Points */
  private_use(0xFE00..0xFFFF),
  (0xFFFF)
} SignatureScheme;
```

```
struct {
  SignatureScheme algorithm;
  opaque signature<0..2^16-1>;
} CertificateVerify;
```

The `CertificateVerify` message contains a signature that serves as proof that the server possesses the corresponding private key for the certificate. This message should always be sent when a `Certificate` message is sent and right before the `Finished` message.

The `algorithm` field contains the signature algorithm used, which belongs to the `SignatureScheme` enumerated. This enumerated belongs to the Signature Algorithms TLS extension. The `signature` field contains the digital signature value processed by using the specified algorithm.

Finished

```
struct {
  opaque verify_data[Hash.length];
} Finished;
```

The `Finished` message includes only the `verify_data` field, which contains an opaque value for the verification hash of the whole handshake. The length of this field depends on the output length for the selected hash algorithm.

Alerts

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
  close_notify(0),
  unexpected_message(10),
  bad_record_mac(20),
  record_overflow(22),
  handshake_failure(40),
  bad_certificate(42),
  unsupported_certificate(43),
  certificate_revoked(44),
  certificate_expired(45),
  certificate_unknown(46),
  illegal_parameter(47),
  unknown_ca(48),
  access_denied(49),
  decode_error(50),
  decrypt_error(51),
  protocol_version(70),
  insufficient_security(71),
  internal_error(80),
  inappropriate_fallback(86),
  user_canceled(90),
```

```
  missing_extension(109),
  unsupported_extension(110),
  unrecognized_name(112),
  bad_certificate_status_response(113),
  unknown_psk_identity(115),
  certificate_required(116),
  no_application_protocol(120),
  (255)
} AlertDescription;

struct {
AlertLevel level;
AlertDescription description;
} Alert;
```

`Alert` messages are sent any time during a TLS session to notify the other end of closure information and errors. These messages contain a legacy field `AlertLevel` used in previous versions of TLS to indicate the severity level of the alert but ignored in TLS 1.3, and a self-explanatory `AlertDescription` field. Alert messages can be categorized as either closure alerts or error alerts, where the former indicates proper closure of the communication in one direction and the latter indicates an abortive closure caused by an application failure. Shared security parameters that were established in a connection that failed must be forgotten by both ends.

Examples of closure alerts include: `close_notify`, which notifies the recipient that no more data will be sent from the other end, causing any additional future data to be ignored; and `user_canceled`, which serves as a notification for handshake cancellation for a reason other than a failure and should be followed by a `close_notify` alert. Error alerts, to name a few, include `handshake_failure`, when the handshake negotiation was not possible with the received security parameters; `bad_record_mac`, when a MAC value in a message is invalid, suggesting that the network might be under attack; `illegal_parameter`, when a message complies with the protocol format but is incorrect or inconsistent in relation to other handshake fields; and `missing_extension`, when a handshake message is received without an extension that is required for the negotiated parameters.

2.3 TLS Vulnerabilities and Attacks

The underlying cryptographic primitives used in TLS protocols are thoroughly validated before publication and finding vulnerabilities in them is relatively rare. Most vulnerabilities for protocols in TLS are actually found in their implementations, as different vendors develop their own versions of the protocol sometimes without paying much attention to security [54, pp. 153-186]. This subsection shows some examples of previously known attacks and vulnerabilities that have been found and exploited in the TLS protocol and its implementations.

2.3.1 Insecure Randomness.

The TLS protocol makes use of different functions for generating random values, e.g., for generating derived cryptographic keys or for creating nonces. Random Number Generators (RNG) need to comply with certain requirements in order to provide enough security in

network protocols [62]. True randomness is really hard to produce with current technology (if not impossible [63]) so cryptographic protocols rely on Pseudo-Random Number Generators (PRNG) instead [64]. PRNGs take information from the system, such as timestamps, CPU metrics, or video and audio inputs, and use them as entropy source for generating seemingly random data.

If an attacker is able to guess or predict future values for random numbers used by either the client or server, the security of the communication can be severely compromised. In 1996, a couple of PhD students found that the now discontinued Netscape Navigator browser was using a timestamp and the process IDs of the browser and its parent in the seed for all SSL random values [65]. They showed that these values do not provide enough randomness and that a skilled attacker with enough information or enough computing power could recover the encryption keys and decrypt SSL messages. Later, in 2008, a vulnerability was discovered in the OpenSSL package developed by the Debian Project, which showed that the implemented PRNG was predictable, thus compromising keys for SSH, OpenVPN, DNSSEC, X.509 certificates, and TLS/SSL sessions [66].

Additional examples of insecure PRNG implementations include: a paper published in 2007 showed that the *CryptGenRandom* PRNG function included in the Windows 2000 and Windows XP operating systems was using a combination of insecure ciphers, seeds and configurations making it vulnerable to key-recovery attacks [67]. NetBSD published a security advisory warning about weak kernel versions with an insecure PRNG function that generated numbers with insufficient randomness caused by a programming error, impacting security [68]. A study of RSA and DSA keys in 2012 showed that insufficient randomness was leading several network devices to share weak keys and to create keys that shared prime factors, which allowed the researchers to recover private keys after obtaining their public counterparts [69].

2.3.2 Heartbleed.

Heartbleed [4], as mentioned earlier, was a vulnerability discovered in the TLS Heartbeat extension [5]. This extension allowed the use of a functionality called *keep-alive* to verify the availability of the parties in a TLS session. This functionality was mainly targeted for the Datagram Transport Layer Security (DTLS) protocol [70, 71], which was meant to provide a security layer similar as TLS over transport protocols considered to be unreliable, such as the User Datagram Protocol (UDP). Since the transport protocols used by DTLS provide no session management, DTLS required a renegotiation from the parties to confirm that any of them was still available. The Heartbeat extension helped the parties confirm the availability for one another without the need of a renegotiation.

The Heartbleed vulnerability was found in the OpenSSL package, which had a vulnerable implementation of the Heartbeat extension that was enabled by default even in Transmission Control Protocol (TCP) messages [72]. This implementation allowed any of the parties to send a Heartbeat request to the other with two parameters: length and payload. The receiver would then read the payload and store the indicated length of bytes in memory with the intention of sending the same payload as a confirmation of network presence. The issue happened when the sent length was greater than the actual size of the payload, in which case the receiver would sent the extra bytes from whatever data it had in memory at that time. This allowed an attacker to get memory maps from the victim in chunks of 64 kilobytes at a time, but by sending several messages, huge amounts of information could be retrieved. This is illustrated in Figure 6.

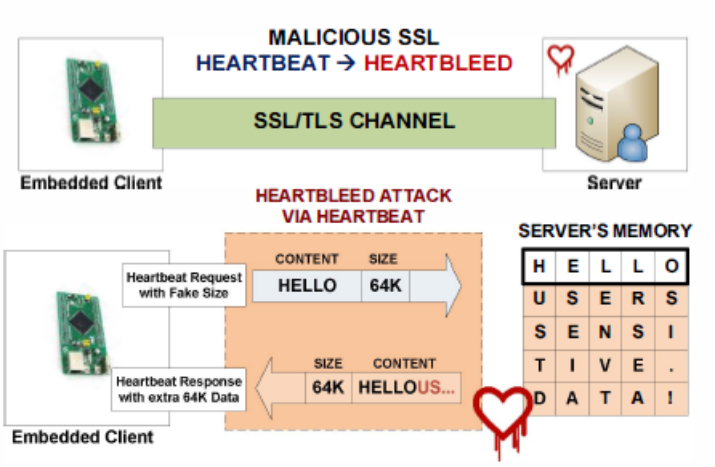

Figure 6: The Heartbleed bug. Adapted from [72]

Depending on what data a server had on memory in a particular moment, a successful exploitation of the Heartbleed vulnerability could have provided an attacker with sensitive valuable information, such as private keys, session tokens, or passwords [54, pp. 164-167]. This data could have been used to steal accounts, hijack user sessions, or decrypt encrypted traffic, causing even more damage. Embedded systems were of particular concern as they often have more limited memory resources and this attack could result in full memory exposure [72]. After its detection, patches were rapidly released and most websites managed to mitigate the issue, though even months later there were still cases of successful exploitations in vulnerable servers.

2.3.3 Bleichenbacher's attack and ROBOT

In 1998, Bleichenbacher published an adaptive Chosen-Ciphertext Attack (CCA) on RSA PKCS #1 that allowed an attacker to decrypt or sign arbitrary messages [73]. The attack requires the existence of an oracle that can distinguish between ciphertexts that have the correct or incorrect RSA PKCS #1 format for the attacker to query. In the attack, the oracle is a vulnerable server and the attacker sends successive modified versions of a valid encrypted message that was previously intercepted. If the modified ciphertext happens to have a valid format, the server responds differently as when the ciphertext is not PKCS #1 conforming, allowing an attacker to distinguish between valid and invalid ciphertexts.

The attack takes advantage of the *homomorphic multiplicative property* of RSA, which makes a CCA attack on plain RSA possible [73]. Roughly, given a message m, ciphertext c of m, with a public key e and private key d that belong to a victim, where the relationship between c and m is given by

$$c = m^e \mod n$$
$$m = c^d \mod n$$

for a given n complying with the RSA algorithm (i.e., $n = p * q$ for sufficiently large primes p and q), then, an attacker can chose a random integer s and calculate

$$c' = s^e c \mod n$$

and ask for the decryption of c'. This decryption is then $m' = (c')^d$, which the attacker can use to recover the original message m by computing

$$m = m' s^{-1} \mod n$$

Since $m's^{-1} = (c')^d s^{-1} = (s^e c)^d s^{-1} = (ms)s^{-1} = m$ modulo p.

PKCS #1 uses an encoding header that starts with two constant bytes (0x00, 0x02 for encryption), followed by a padding string of varying length but of at least 8 bytes, followed by a constant zero byte separator, and finally by the encrypted message. In Bleichenbacher's attack, the attacker intercepts a message c and generates a value s to generate c' as stated above, which is then sent to the oracle or server. The server attempts to decrypt the message, obtaining m', and responds differently according to whether the first two constant bytes are found in m' or not. The process is repeated with different values for s until a valid m' is found, which happens with a non-negligible probability.

Once a valid s is found, the attacker knows that ms is in a specific range, which can be used to further guide the queries until the value of m itself is found or the range is small enough to be brute-forced. At this point, the attacker has been able to find the decrypted value of the encrypted intercepted message without access to the private key. The attack is said to be *adaptive* since late queries are forged based on the results obtained from previous queries.

In order to mitigate this attack, some countermeasures were proposed. One of them was to make servers respond with generic messages to any query so that attackers could not deduce when a forged ciphertext complied with the format or not.

The Return Of Bleichenbacher's Oracle Threat (ROBOT) [14] attack was published almost 20 years later, in 2018, and it showed that a large amount of servers were still vulnerable to Bleichenbacher's attack and some of its variations. In their publication, Böck et al. managed to find several vulnerable servers belonging to well-known companies (such as, Cisco and IBM) and vulnerable open source implementations after performing a large-scale scanning and they even obtained the private key of Facebook. They found that by modifying the TLS protocol flow, they could still distinguish between the responses sent by the servers when a valid or invalid message was sent, which allowed them to use the server as an oracle and bypass previous mitigations.

TLS 1.3 removes RSA altogether so, in theory, it is not vulnerable to any of the Bleichenbacher-based attacks. However, researchers found that if previous versions are enabled in a server for compatibility compliance, TLS 1.3 is not enough to protect against these attacks [74].

2.3.4 CRIME

The Compression Ratio Info-leak Mass Exploitation/Made Easy (CRIME) attack [75] was published in 2012 and attracted significant attention from the security community since it abused the compression mechanisms implemented in TLS versions 1.2 and below to leak information. The attack was developed by the same researchers that, just a year earlier, published the Browser Exploit Against SSL/TLS (BEAST) attack [76], which exploited vulnerable implementations of the AES algorithm in CBC mode for TLS 1.0 and below to steal session cookies.

In a CRIME attack, the attacker has packet sniffing capabilities on the victim's channel and a malicious or compromised server that waits for a victim to connect. Once connected, the server uses scripts to force the victim browser to make requests to a target website for which the victim has a valid session. The forged requests contain data generated by the attacker as well as from the browser which is compressed together. The attacker then eavesdrops on the encrypted packets sent by the victim, taking note on the size of the ciphertexts, and sends different requests changing the destination path which she controls. When the destination path coincides with some portion of the browser-generated data, the compression algorithm reduces the size of the packets, which the attacker notices through sniffing. By carefully sending modified contents and looking at the traffic, the attacker can infer parts of the plain text data, with the goal of obtaining the victim's session cookie.

The best solution found for compression vulnerabilities like CRIME was to disable TLS-level compression altogether. This was implemented by most browsers at the time and TLS 1.3 completely removed it from the specification for the same reason.

2.4 Fuzzing

Fuzzing is a widely used technique for finding bugs by automatically generating valid and invalid data and feeding it into software applications while monitoring their behavior [11]. Fuzzers generate large amounts of test cases out of pre-configured templates/rules or by mutating valid data, which are then used to try and reach as many portions of the code as possible, hoping to trigger software errors, crashes or memory leaks. Once an error is found, the test case can be reviewed manually to find the potential causes.

The concept of testing applications with random inputs is not new. Duran et al. [77] published in 1981 a report on the effectiveness of random testing with positive results, e.g., it can be cost-effective, since complex bugs are actually found with relatively low effort. However, the term "fuzz" was not used until a publication in 1990 where Miller et al. [78] coded a fuzzer to test UNIX utilities with random data. Ever since it was first introduced as a software testing method, new improvements have been made for better efficiency and speed, and it has gained increased popularity [79, 80].

Several security bugs have been found using fuzzing tools. These tools are often combined together to obtain a wider insight into the vulnerabilities present in software applications. Fuzzing tools, such as Honggfuzz [81], AFL [82], and OSS-Fuzz [83] have been used to successfully find security vulnerabilities in popular software libraries and packages, such as OpenSSL and Apache. Also, it has been shown that previously discovered vulnerabilities could have been found through fuzzing techniques if used properly, which is the case for the Heartbleed vulnerability, for example, which was not originally found in this manner [84, 85].

2.4.1 Fuzzing process and components

The process of fuzzing a specific target application or Program Under Test (PUT) based on some correctness policy or to find software errors, also known as a *fuzz campaign* [79], involves different stages from input generation to bug detection. Liang et al. [11] define a set of components in which the fuzzing process can be divided. Most fuzzers include these components either as software modules or instruction blocks, or leave them as manual work for the tester. The components, as illustrated in Figure 7, are explained below.

In fuzzing, higher code coverage degrees are considered signs of efficiency for a fuzzer, as it can be used to measure how much of the attack surface of the application is reached by the tests [86, 130-133]. Because of this, some fuzzers use coverage to guide their input generation (e.g., [81, 82]), i.e., in order to generate more appropriate test cases, these fuzzers need to monitor the behavior of the program as they input different data. This can be done using a built-in *monitor* component that checks the runtime information from the program. Such monitor can also take advantage of techniques, such as *code instrumentation*[87], in which additional instructions are injected into the program (either in the source code, at runtime, compile time or post link time) to observe its behavior at different stages; or *taint analysis*[88], in which patterns of instructions that are previously known to be dangerous are used to detect untrusted or tainted portions of code. Other fuzzers, as discussed below in Section 2.4.2, do not bother in obtaining any internal information from the PUT for guiding the generation of test cases. For these fuzzers, the execution flow is done blindly, so a monitoring component is not required.

A *bug detector* can be a portion of code or function in the fuzzer that keeps track of crashes and error reports in the target program, while collecting relevant information. The alerts generated by the bug detector can then be filtered or manually reviewed to distinguish real issues from false positives. Often, a combination of code analysis, debugging tools and memory error detectors, such as AddressSanitizer [89] or MemorySanitizer [90] from Clang, can be used to further investigate potential holes or find more hidden bugs.

The *test case generator* creates the new test cases that will be used as input for the PUT, either by changing seeds or using known grammar as explained in Section 2.4.3, for which it may process the information gathered by the *monitor* and use it to guide the generation, if applicable. The *bug filter* refers to the manual process of selecting which of the reported bugs are actually exploitable or relevant, discarding false positives and negatives. Even though researchers have come up with novel ways to do a preliminary filtering of bugs to reduce the amount of manual work [91], this still remains as a task mostly done by the tester.

Overall, the fuzzing process is as follows:

1. The tester identifies the target application or PUT and the format of the data to be used as input. The tester needs to take into account what changes need to be made to the mutated data depending on the format of the data that the application accepts. For library calls, for example, it might be necessary to create an additional program that formats the input data and calls the respective library functions to be fuzzed. This program is generally known as a *harness*.

2. The files to be used as corpus are generated/collected and fed into the fuzzer, which will use them to generate new test cases using methods as previously explained.

3. The fuzzer runs the target with the generated test cases for a set of iterations or for an indefinite amount of time. During this phase, new test cases might be generated depending on the behavior of the target program.

4. The program is monitored and analyzed to look for crashes, errors, or other interesting activities.

5. The tester analyzes the results and discards data that is not useful or tries to obtain more information regarding potential bugs. This step is often done manually.

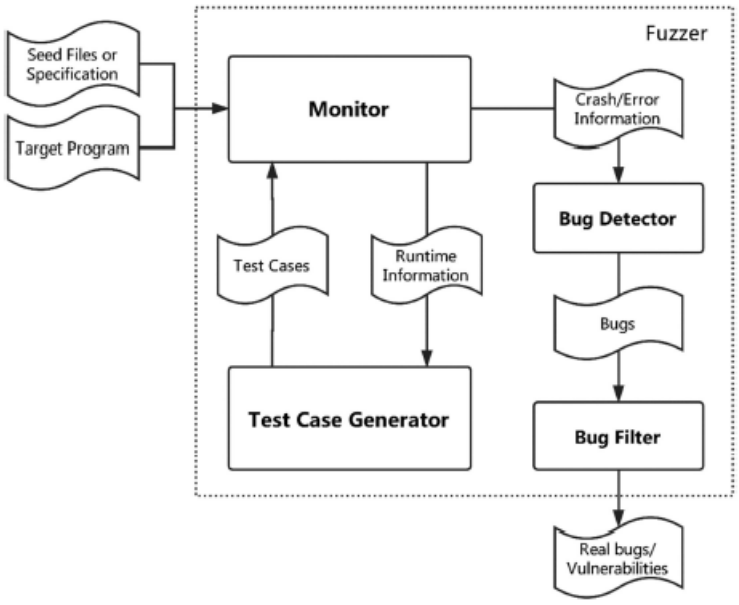

Figure 7: General fuzzing process. Adapted from [11]

2.4.2 Fuzzer classifications

Similar to regular software tests [92, p. 224], fuzzing approaches can be classified in the following three categories based on the knowledge available on the PUT [79, 11]:

Black-box Fuzzing. The fuzzer has access to only the inputs and outputs of the PUT, with no access to internal primitives or code. The fuzzer takes the different responses and outputs of the program into account when finding issues by comparing them with varying criteria, depending on the fuzzer, to assess the test verdict [86, pp. 83-86]. Nonetheless, the fuzzer can still get information about the format or "grammar" required for the input in order to generate semi-valid cases. A black-box fuzzer has the advantages that it is commonly faster than others and is easily compatible, although the generated test cases tend to provide low code coverage and there are higher chances of missing hard-to-reach functions or branches [11]. Some examples of black-box fuzzers include Funfuzz [93], Wfuzz [94], and Trinity [95].

White-box Fuzzing. The fuzzer is assumed to have complete knowledge of the PUT internals and behavior and can create the test cases based on this information. This approach was initially proposed by Godefroid et al. in 2007 [96] by using a method they called Dynamic Symbolic Execution (DSE), a variant of symbolic execution [97], and coverage-based search algorithms. The information obtained from the PUT is then used to further guide the generation of test cases.

DSE works by executing the PUT with some initial test cases and then looking at all the conditional statements in the program that would trigger an execution switch or branch, depending on whether the condition is met or not. These branching behaviors generate a constraint when the condition is not met and the branch does not occur, skipping a

block of instructions that would have been executed otherwise. A constraint solver is used to infer what data can be generated to pass the condition and force the execution of the skipped block. This way, the fuzzer uses information from the application to guide the generation of test cases so that they meet the requirements and reach as many execution paths as possible.

White-box fuzzers can theoretically generate test cases with better quality or that can trigger more interesting behaviors. However, this constant behavior monitoring introduces a higher overhead when compared to black-box fuzzers, makes scalability harder and can bring compatibility issues. Furthermore, the large amounts of possible execution paths in real software systems make solving constraints much more complex in practice [11, 79]. White-box fuzzer examples include SAGE [98], BitFuzz [99], and MutaGen [100].

Gray-box Fuzzing. A combination of both white-box and black-box fuzzing where the fuzzer gets some partial information on the PUT (e.g., by doing some lightweight static analysis of the PUT) and also modifies the test cases based on dynamic information obtained at runtime or instrumentation that generates coverage information. Although similar, white-box and gray-box fuzzers differ mostly in that gray-box fuzzers use only some portions of information from the program, such as code coverage, to make decisions on which paths are missing to test, while white-box fuzzing attempts to reach all paths from the beginning using the source code or other deeper information. Examples of gray-box fuzzers include Honggfuzz [81], AFL [82], and LibFuzzer [101].

2.4.3 Test case generation

Fuzzers need either a collection of sample files to use as input, called the *corpus* or *seed*, or a specification that tells them how to generate proper input. Each fuzzer has their own algorithms and strategies for mutating the corpus and generating new test cases in ways that are more likely to trigger additional calls in the PUT and improve the coverage. *Mutation-based* generation happens when the fuzzer makes mutations to a pool of seed files either randomly or by following a set of pre-defined rules or operations that can also be changed after obtaining additional runtime information. On the contrary, *grammar-based* generation occurs when the fuzzer generates data based on a format or specification (called the "grammar"). The generated data is semi-valid, allowing it to pass early parsing and conditions but that will hopefully trigger errors in later stages [11].

A third approach is also occasionally considered in the literature where the fuzzer generates purely random data with no context or previous information whatsoever, as discussed in [88]. This method has been successfully used in the past to find vulnerabilities [78, 102] but, although fast and simple, provides only superficial testing and poor coverage.

Grammar-based (also called model-based) generation can take, for example, protocol specifications or templates that indicate the amount and types of arguments that are expected for a given application. Some implementations also accept grammar lists that include specific keywords known to be recognized by the PUT, which can be useful to test SQL database servers or programming languages, for example. Although not so common, some tools include inferred modelling, where the fuzzer attempts to predict suitable cases based on, e.g., string literals or network captures [79]. Funfuzz [93] and Wfuzz [94] are examples of grammar-based fuzzers.

Mutation-based generation involves applying operations on existing data, such as flipping a fixed or random amount of bits, interpreting byte sequences as integers and

performing basic arithmetic on the value, dividing the data in blocks and working on each block (e.g., by inserting, deleting, replacing, or truncating them), to name a few [79]. This helps increase the probabilities of creating test cases that are similar to the original seed, but that contain subtle differences to trigger new behaviors. Honggfuzz [81] and AFL [82], for example, generate inputs based on mutations.

In addition to these categories, fuzzers have also been classified as being *smart* or *dumb* depending on how "aware" the fuzzer is of the input's structure [86, pp. 142-144]. A *dumb* fuzzer (e.g., AFL [82]) performs mutations at random based on pre-defined rules without considering the meaning of the specific fields or offsets it is changing. A *smart* fuzzer (e.g., [103]) has a better understanding of the meaning of the fields in the inputs and can, for instance, perform changes according to a protocol RFC. Completely dumb fuzzing produces low coverage and might end up stressing the parsing code rather than reaching the main components of a PUT. A fully smart fuzzer, on the other hand, would comply with the specifications or RFC to the maximum, uncovering no bugs at all. Typically, fuzz tests should aim for a level between these two categories.

The testing framework developed for this thesis uses a combination of grammar and mutation-based black-box fuzzing, along with static and functional tests to find bugs in TLS implementations.

3 Target system

In this Section, the system that was used as a target for the tests is described.

3.1 HTLS

HTLS is a TLS 1.3 implementation by HSSL. It is written from scratch and provides a small footprint, with about 60 Kilobytes of code and 10 Kilobytes of used RAM per TLS connection. It is potentially efficient as memory addresses are localised to a small area and it avoids the use of the function *memcpy* when possible, using, for example, back-to-front encoding for TLV data. It is also suitable for Trusted Execution Environments and secure enclaves [15, 104] since it has no dependencies or memory allocations within the library. All these features make it optimal for use in constrained-resources devices, such as for IoT devices and other embedded systems.

At the time of testing, HTLS supports the PSK and ECDHE handshakes for client authentication, the P-256 elliptic curve for ECDSA and ECDHE, HMAC-SHA256 for HKDF and the TLS_AES_128_GCM_SHA_256 cipher suite. In addition, this implementation supports ASN.1 DER encoding and decoding, basic X.509 certificate encoding, decoding and validation, basic support for remote attestation, and the Trusted Sockets Layer (TSL) protocol [20].

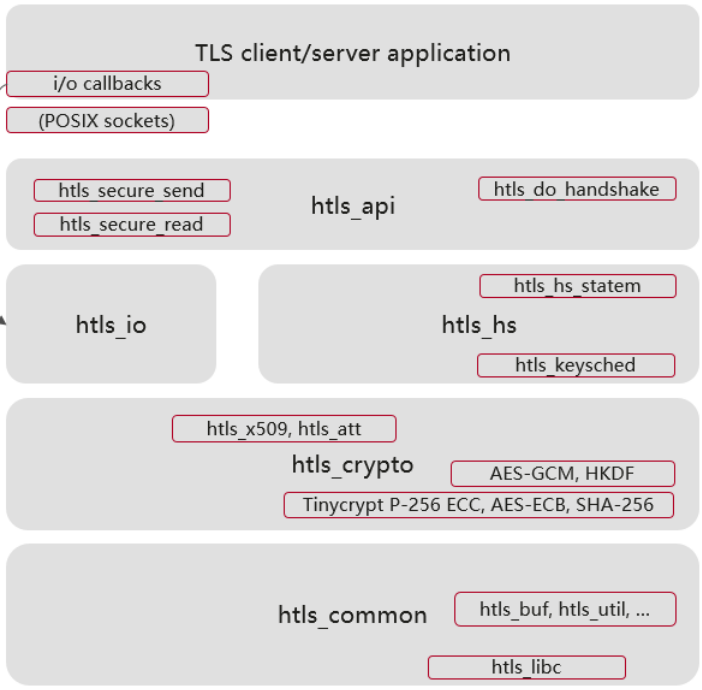

Figure 8: HTLS library components

As illustrated in Figure 8, HTLS is divided into a set of library components that interact with each other from higher to lower levels. These components are the following:

`htls_api.` Contains the top level API for creating a secure connection between TLS peers over the network. It includes high-level functionality to initiate, accept and terminate

a connection, parsed received identities and certificates, perform a TLS handshake, and securely send and receive data over an established channel.

`htls_io`. Includes lower-level functions for sending and receiving buffers of data. It is used by `htls_api` for all peer communication functions. This component is agnostic to the communication mechanism (e.g., sockets or shared memory) and depends on the application to provide communication callbacks that can be invoked by `htls_io` for the actual transmission of data.

`htls_hs`. Contains all handshake related methods, including handshake initialization and parsing, writing and processing of all handshake messages (e.g., `ClientHello`, `ServerHello`, `EncryptedExtensions`, `Certificate`, `CertificateVerify`, `Finished`, `Alert`), parsing of PSK extensions and data related to the negotiation of security parameters, manipulation of the transcript hash (e.g., initialization, updating, generation of signatures), and keeping track of the current handshake state and expected messages to be received from the peer.

`htls_crypto`. Includes all methods related to cryptographic functions and calls Tinycrypt's APIs for some of them, as discussed in the following section. This refers to methods related to HKDF, PRNG, hashing, HMAC generation, AES encryption and decryption, generation of ECC keys and shared secrets, signing and verification of signatures, among others.

`htls_common`. Includes all low level functions used by the previous components, e.g., methods for string manipulations, along with additional low-level library components, such as: `htls_buf`, which is the main component for secure manipulation of buffers; `htls_util`, which includes functions for reading files and data conversion; `htls_types`, which defines global constants; and `htls_libc`, which is a small fingerprint in-house implementation of several libc functions for systems with very constrained resources.

3.2 Tinycrypt

Tinycrypt [18] is a library developed by Intel with the purpose of providing small-footprint cryptographic capabilities for constrained devices. It contains a minimal set of cryptographic algorithms and primitives for providing cryptographic security without the need for large space or RAM consumption. In order to optimize code, some primitives depend on others, allowing for code reuse when possible. At the time of writing, Tinycrypt supports the following primitives:

- SHA-256
- HMAC-SHA256, which depends on SHA-256
- HMAC-PRNG, which depends on SHA-256 and HMAC-SHA256
- AES-128 (ECB)
- AES-CBC, which depends on AES-128
- AES-CTR, which depends on AES-128

- AES-CMAC, which depends on AES-128
- AES-CCM, which depends on AES-128
- CTR-PRNG, which depends on AES-128
- ECC-DH, which depends on ECC auxiliary functions
- ECC-DSA, which depends on ECC auxiliary functions

Tinycrypt was designed with the specific goals of minimizing the space required by the code for each cryptographic primitive and minimizing the dependencies between the primitives. Particularly, it allows developers to choose and build the specific primitives that are required for their applications. With so many code optimizations, Tinycrypt also includes a set of limitations, as mentioned in their repository, that the developers are aware of and accepted in order to fulfill the goals and expected code size. Most of these limitations have low risks or are only present under certain conditions, and developers using the library can implement features in their applications to overcome them. Some of them include: a missing overflow check for hashed bits in the SHA-256 function for more than 2^{64} bits (which is an extremely huge window), lack of support for keys of lengths other than 128 for AES (which is hardly required in constrained devices), and lack of a built-in secure PRNG function for ECDSA key generation or signing (which developers need to add themselves), to name a few.

The ECC implementation in Tinycrypt is based on the micro-ecc project [105], which is an optimized implementation for ECDSA and ECDH written in C. Micro-ecc supports different processor architectures, has very small code size, and supports five of the most used standard curves. An important remark, is that uncompressed points are represented in the standard format with the difference that the initial prefix byte 0x04 is omitted and it is left for the developers to make sure this is taken into account before calling ECC functions. HTLS removes this byte when present to avoid conflicts.

HTLS makes use of Tinycrypt as the low-level implementation for some of the cryptographic primitives it supports, namely ECDSA and ECDH in the P-256 curve, AES-ECB, and SHA-256. On top of Tinycrypt, HTLS provides its own implementations of GCM, HKDF, and ASN.1 DER encoding and decoding. Additionally, the micro-ecc project (and consequently both Tinycrypt and HTLS also) uses Shamir's trick discussed in Section 2.1.9 to optimize the ECDSA signature verification as well as the *(X, Y)-only* variants of the co-Z formulae for point additions discussed in Section 2.1.8.

4 Testing framework design

This section explains how our testing framework for the target system was designed and implemented. At the beginning of the research project, HTLS already included basic static unit tests and interoperability tests, which are now part of the framework. In addition, a combination of static tests and fuzzing techniques was used to cover as much of the attack surface as possible. Fuzz testing was the main approach we used for vulnerability discovery, however, there are portions of the implementation that are very unlikely to be reached with fuzzing alone. Therefore, hand-crafted test vectors, like those included in the Wycheproof project discussed below, were also used.

4.1 Tools

This subsection details the tools that were considered for integration into the testing framework and how they were included to contribute to the tests. These tools are all public and written by third parties.

4.1.1 Wycheproof project

The Wycheproof project [17] provides a set of tests for cryptographic libraries designed and developed by the Google Security Team. The goal of the project is to compile a collection of tests for third party cryptographic software libraries that can be used to measure their security and find weaknesses. It contains both algorithms that test different portions of cryptographic primitives as well as a wide array of single use test cases with configurations and formats known to break or trigger unexpected behaviors in similar software components.

Wycheproof includes security tests for several cryptographic algorithms, out of which the ones that are particularly interesting for this thesis are AES-GCM, ECDH and ECDSA. Most tests are based on relevant scientific research and consider scenarios and cases of varying complexity. However, as stated in their own repository documentation, passing the Wycheproof tests does not necessarily mean that a library is secure, rather it shows that there are sufficient controls implemented to mitigate the specific tests it has.

The testing framework designed for this thesis includes all the relevant static test cases for the algorithms implemented by the target system. These test cases were taken from the Wycheproof project and fed to the relevant function in the target to measure their behavior by creating a harness similar to those used in fuzzing. This helped to find potential weaknesses and errors that were later fixed. The static tests were included and compiled in scripts to allow for a single command to be executed that would show which specific test cases failed.

4.1.2 Honggfuzz

Hongfuzz [81] is a gray-box multi-threaded security-oriented fuzzer developed by Google that has been around since 2010. It has been widely used by researchers and developers to find vulnerabilities and bugs. Some of the software packages where Hongfuzz has been used to find security issues include Apache, OpenSSL, LibreSSL, Adobe, and others. It has also been extensively integrated or used as a baseline for several fuzzing tools and projects, a list of which can be found in their repository page.

Every time Hongfuzz detects that a crash occurs, it saves the payload that triggered the crash in the same location as the seeds, so it is reused as a new corpus in the mutation algorithm to generate new test cases that are more likely to trigger similar crashes.

Hongfuzz has two main modes:

Feedback-driven mode. This mode utilizes code coverage to guide the fuzzing process. By using hardware-based (e.g., cpu branch counting, Intel branch and processor tracing, etc.), sanitizer coverage (which includes instrumentation points in user-defined functions for fuzzing) and/or compile-time instrumentation, Hongfuzz is able to find what inputs add new coverage paths and then add them to the corpus to be reused. Hongfuzz then keeps mutating the corpus as it changes through new fuzzing rounds and looking for payloads that trigger new coverage.

Persistent mode. This mode allows Hongfuzz to test APIs, but it requires a software harness for testing the specific APIs or functions. A harness is a binary that imports the library and calls its functions directly, using the mutated data generated by Hongfuzz as input. This method is commonly used by other fuzzers.

The Feedback-driven mode works best for applications that process data from external files or from stdin, but the fuzzed functions are limited by the calls done by the application when reading this external data, which is the reason why measuring the coverage makes sense for improving the results. With this fuzzing, some functions might be left out as they will never be reached by the mutations, and that is when the persistent mode is useful. The best approach is, of course, a combination of both so that the code is tested as widely as possible.

In addition to these two modes, Hongfuzz provides the capability to do black-box fuzzing of binaries through the QEMU emulation software, as well as some tools for fuzzing through sockets, allowing to perform fuzz tests over the network. Unfortunately, these two features are not very well documented and require a big amount of reading through their code in order to understand how to properly use them.

Our testing framework uses a combination of the feedback-driven and persistent modes to test some portions of the target system. Sadly, given that cryptographic protocol implementations mostly depend on several variables that are set during a session between two peers (e.g., TLS needs to know the handshake state at all times as well as the negotiated cipher suites to use), obtaining high code coverage while fuzzing with tools like Honggfuzz is significantly hard. We encountered issues when trying to simulate these variables with static harness binaries and file-based fuzzing worked only with standalone binaries that performed specific and straightforward functions, such as validating signatures or parsing public keys, thus the coverage we obtained was generally low.

4.1.3 AFL

The American Fuzzy Lop (AFL) [82] tool is a newer gray-box fuzzer also developed by Google known to be very powerful and to have found several remote code execution and privilege escalation issues in software libraries. Similarly to Honggfuzz, AFL uses instrumentation to guide the fuzzing in its default mode and reuses test cases that were found to reach new state transitions. By contrast, AFL attempts to optimize the used test

cases by trimming them to the smallest size possible trying not to change the behavior of the application.

Applications that read data from a file or stdin can be fuzzed right away after being compiled with the required instrumentation, while specific API and library calls can be added by constructing a code harness. Just like Honggfuzz, AFL supports black-box fuzzing when the source code is not available via QEMU.

In comparison to Honggfuzz, AFL has more complete documentation and online resources written by the community, making it mostly easier to use and adapt to different scenarios. In addition, it includes a set of binaries that help with result analysis and test case minimization. This tool was also adopted in our testing framework in order to improve our harness and file-based fuzz tests, however, we had the same difficulties as with Honggfuzz and did not manage to obtain high code coverage.

4.1.4 AFLNet

Both AFL and Honggfuzz are great tools to fuzz specific low-level functions from the library implementation which are used during a TLS session, but both fall short for testing protocols over the network. Since the TLS protocol behaves differently according to what data is included in each of the handshake messages, there is a need for a tool that can test the protocol by mutating each of these messages. In addition, in order to test late messages, a valid handshake state needs to be reached first by sending a set of valid messages. Traditional fuzzers are generally not capable of sending mutated test cases via socket connections and when this functionality is indeed included, it tends to be hard to use and/or poorly documented.

AFLNet [106] is a gray-box fuzzer that attempts to solve the challenges faced by other fuzzers when testing network protocols, such as the inability to fuzz through sockets, to include the session state for guiding test case generation, or to use a sequence of messages out of which some are valid and should not be mutated. AFLNet is based on coverage and was implemented as an extension to AFL, adding support for socket communication. Broadly, AFLNet works by reading packet capture files which contain requests and responses exchanged between a client and server communicating over the protocol to be fuzzed, e.g., FTP. Then, it parses the protocol-specific message sequences based on built-in rules and produces an initial corpus for generation of test cases. Finally, it continues the regular fuzzing process by automatically generating new cases through seed mutation and sending them over sockets to the PUT, while keeping track of the states and coverage.

AFLNet includes a new learning algorithm that notes state changes for particular test cases based on the responses sent by the server. This, in combination to AFL's mutators and coverage guidance, helped the tool outperform the gray-box fuzzer BooFuzz [107]. Although initially developed to fuzz the protocols FTP and RSTP, support for additional protocols was added later, e.g., for DNS, SMTP, SIP, and TLS.

Even when AFLNet produced positive results when fuzzing the FTP and RTSP protocols, it gave poor performance when we used it to fuzz our target TLS implementation. After feeding it a capture file with valid `ClientHello` messages, AFLNet's mutation algorithms failed to initiate a TLS handshake, making data modifications on short TCP packets instead of the seeds and generating packets that were not able to reach the TLS layer at all. As explained in their documentation, these difficulties could have been fixed by making code modifications in the server, which is a troublesome solution. Furthermore, AFLNet fails to include cryptographic capabilities that are required to fuzz late handshake

messages, e.g., encryption and decryption of messages, MAC integrity validation checks, etc. Therefore, it is still not a suitable solution to fuzz cryptographic protocols.

Overall, implementing both Honggfuzz and AFL into our testing framework proved to be difficult mostly due to the complexity of the target library methods when building harnesses, while AFLNET failed at TLS fuzzing. Honggfuzz and AFL work great with binaries that take input from files, but a network implementation such as HTLS makes harness development mandatory. Furthermore, higher-level functions that would reach a wider code surface are dependent on context-aware variables, such as the state or transcript of a particular handshake, which are highly difficult to simulate in a static harness. This resulted in low-coverage harnesses for specific functions where not many execution paths were found. When the harnesses were tweaked properly, both these fuzzers managed to find some of the less complex bugs that had already been found by eGMT, described in Section 4.2.3, but overall they did not offer an efficient performance for our specific target scenario.

Based on the results obtained from these initial tests, we decided that a better approach would be to develop our own fuzzing method based on previous research. The following sections discuss the fuzzing techniques considered for the development and design of eGMT. Figure 9 displays a summary of the fuzzers mentioned previously.

Name	Classification	Input Generation	Main Advantage	Main Disadvantage
Honggfuzz	Gray-box	Mutation-based	Well studied	Hard to reach high coverage through harnesses
AFL	Gray-box	Mutation-based	Highly documented	Hard to reach high coverage through harnesses
AFLNET	Gray-box	Mutation-based	Supports network fuzzing	Not very effective with cryptographic protocols

Figure 9: Summary of the fuzzers considered for the testing framework

4.2 Fuzzing techniques

4.2.1 TLS-Attacker

TLS-Attacker [12] is a security testing framework written in Java used for evaluating TLS libraries. It works by modifying TLS messages and protocol flows, and sending them to a TLS server to test the server's implemented countermeasures against cryptographic attacks and find vulnerabilities. In the original work [12], it was successfully used to find security issues in well-known TLS libraries, such as OpenSSL and GnuTLS. TLS-Attacker provides a handful of binary demo applications that can be used out-of-the-box to test for the presence of specific types of vulnerabilities in TLS implementations. However, Somorovsky, the author, intended for the framework to be preferably used as a programming library, allowing for more powerful functionalities.

The TLS-Attacker framework introduces the concept of "modifiable variables", which act as wrappers for numerical, string and byte types in TLS handshake messages and

records, and allow to map values to specific message fields to later facilitate modifications as needed. These variables also keep track of the original value (before mutation), aiding in manual analysis when bugs are discovered. This can be used to manually create a set of static tests to be sent to a server or integrated into an automated process for dynamic fuzzing. Some of the modifications included in TLS-Attacker for modifiable variables are:

- **Explicit value.** Set a specific value explicitly for an integer, byte array or string.
- **Add.** Add a number to an integer or single byte field.
- **Subtract.** Subtract a number from an integer or single byte field.
- **Right/left shift.** Perform a bitwise shift to an integer or byte array in the right or left direction for a specific amount of bits.
- **XOR.** Perform a bitwise XOR operation on an integer or byte array with another specific value of the same type.
- **Insert/delete.** Perform an insertion or deletion of bytes at a specific position in a byte array.
- **Shuffle.** Shuffle the bytes contained in a byte array.
- **Duplicate.** Duplicate the bytes in a byte array.

Additionally, TLS-Attacker was developed with TLS functionality for derivation of keys and encryption/decryption support, along with message handlers for each of the messages in the protocol. Somorovsky also included an XML module to ease the creation of custom protocol flows for non-Java developers. These flows can be defined in single XML configuration files that indicate what message order to follow and some of the values to be included in the messages, e.g., what cipher suites to include in a `ClientHello` message. Although the framework was initially developed for testing TLS 1.2 and versions below, support for TLS 1.3 was added later in 2018 and the framework has been improved continuously, having the latest contribution at the time of writing in 2021.

Somorovsky used the library to develop a set of tests to measure the effectiveness of the framework by using it to assess the security of common TLS libraries. The tests were organized in two stages. The first stage would check for the possibility of executing some cryptographic attacks, namely, Bleichenbacher's attacks (discussed in Section 2.3.3), padding oracles [108], POODLE attacks [109] and invalid curve attacks [110], all of which are included in the demo binaries.

The second stage used the modifiable variables on a set of correct and incorrect TLS protocol flows to create mutated versions of the flows and fuzz a TLS server. The fuzzer would first try to send the correct and incorrect flows without modifications, while storing the flows that were correctly executed. The incorrect flows were taken from previous research [111, 112, 113] where it was found that such invalid data could trigger bugs in TLS libraries, so a correct execution of the flow (i.e., completing the flow to the end) would already present a problem. Then, the flows that executed until completion would be mutated to further fuzz the server.

The mutations would be done sequentially on each and all of the modifiable variables mapped to the message fields in order to look for specific variables that influenced the correct validation of the protocol flow. This was done since some fields did not seem to

cause any changes after modification (e.g., the `random` field in the `ClientHello` message) and the author considered best to not modify them at all. Once these "influential" variables were found, they were mutated randomly by applying mutations relevant to their type, such as the ones listed earlier, generating modified versions of the original protocol flows. As a final group of tests, the fuzzer would create randomized protocol flows by adding or removing random messages from the original flows.

In the original paper, this approach helped to find several bugs in different TLS libraries. However, the fuzzing steps initially created by Somorovsky in the second stage are no longer included in the project and, unfortunately, the relevant documentation is scarce and brief. Implementing a fuzzer using the TLS-Attacker framework now requires a huge amount of code review and can be time-consuming, especially for non-Java developers. Furthermore, most of the attacks included in the demo binaries are no longer relevant for TLS 1.3 (e.g., Bleichenbacher's attacks target RSA and padding oracles abuse errors in CBC cipher suites, both of which are not supported by TLS 1.3). For these reasons, we opted not to integrate the framework in our tests, but rather take ideas to improve our fuzzer, eGMT, discussed further in Section 4.2.3.

4.2.2 GMT-based Differential Fuzzing

Walz and Sikora proposed in [13] a new black-box fuzzing approach based on *Generic Message Trees* (GMTs) for testing TLS implementations, simply called "GMT" for the rest of this thesis. In their work, they found that this approach caused more behavior variations in comparison to other tools, such as AFL and TLS-Attacker [12]. For this comparison, they first generated a set of inputs that were then fed into five TLS implementations to note any differences in their executions, a technique known as "differential fuzzing". The main target for GMT-based fuzzing is the TLS handshake and more specifically fuzzing the ClientHello message. Even though the GMT-based approach can be used for other handshake messages, the need of encrypting and decrypting these messages as well as generating valid HMACs adds complexity to the implementation of such tests and thus, was left as future work by the researchers.

Walz and Sikora use the TLS Presentation Language (TPL), discussed in Section 2.2.2, to generate parse trees from valid data where each node is a protocol message field. Given that TPL definitions are not sufficient on their own to map specific fields to GMTs automatically, they make use of an enhanced version of TPL which they implemented in previous research [114]. As illustrated in Figure 10, the raw bytes of a ClientHello message are read and converted into a GMT representation. The nodes in the GMT are then iteratively subjected to different operations to generate valid and invalid versions of the message, which can then be sent to a TLS server for fuzzing tests.

The approach developed by Walz and Sikora works broadly as follows:

1. The GMT representation of a valid `ClientHello` message is obtained and used as a baseline for generation of test cases
2. A random target node in the GMT and a random operator are selected
3. The operator is applied to the target node changing, deleting or adding data as applies
4. If applicable, the repairing operator is called, which randomly decides whether to fix

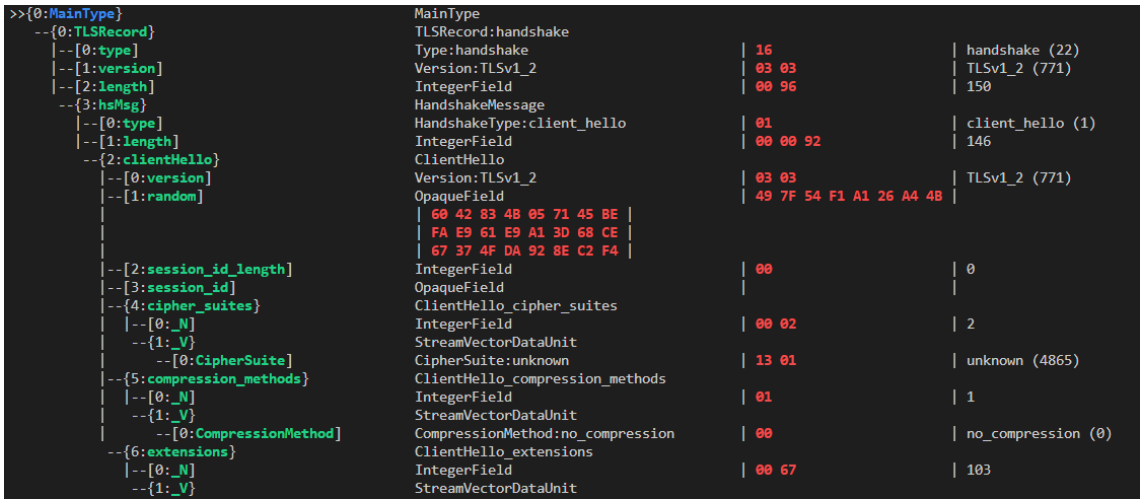

Figure 10: GMT representation of a raw ClientHello message.

the length fields in the object from the target node up to the root level. There is also a random option to do a full repair of all lengths

5. Steps 2 through 4 are repeated for a random number of times

6. The final object is encoded back to bytes and saved in a file to be used as fuzzing input. This fuzzing is done by a separate script

It is evident that this approach relies heavily in proper randomization. Walz and Sikora noticed that there would still be cases where after all operations are completed for a particular iteration, the resulting test vectors are not always distinct. To deal with these cases, they hashed each of the resulting byte strings and kept a list of the hashes. When a hash matches another one in the list, indicating the test vector was already generated before, it is discarded.

A total of 8 different operators are used by GMT to alter the bytes in the `ClientHello` message. An additional repairing method selects at random whether to fix all or certain length fields in the mutated tree, bringing the possibility of having valid lengths or intentionally having cases with length fields that are not consistent with the actual data field. This helps to measure how the application reacts to such inconsistencies. The node operators as defined in [13] are the following:

Voiding Operator. Replaces a node or subtree with empty data while keeping a place marker where the original data was before. This allows for subsequent operations to be applied in the same position, even if no data exists any more.

Deleting Operator. Deletes a node or subtree without including a place marker. Subsequent operations will not be able to affect this position.

Duplicating Operator. Duplicates a given node or subtree and adds it in the GMT as a sibling of the original.

Truncation Fuzz Operator. Chooses a new random length lower than the original length for a subtree and truncates the data in the subtree so that it has the new length, removing child nodes and subtrees as necessary.

Fuzz Int Operator. Set a node to a new random integer. This operator has the option to use what the researchers call "proximity mode", in which the new integer is chosen between 0 and $2 * x - 1$, where x is the integer representation of the original value. Otherwise, the "full range mode" is used, where the new integer is chosen between 0 and the maximum value allowed for the corresponding amount of bytes in the stream. Whether to use either of the two modes is decided at random with a probability of $p = 1/2$.

Fuzz Data Operator. Replaces a node with new random data. It is very similar to the *Fuzz Integer Operator* with the difference that it does not include the "proximity" and "full range" modes. This operator has the capability to select a new length for the node (with a maximum possible value of $2 * l + 1$, with l being the previous length) and resize the node as new data is generated. However, the code for GMT has a hard-coded option to always preserve the original length, so the resize option is not really used.

Appending Fuzz Operator. Generates a new random node of 1 to 4 bytes and appends it to a subtree as a sibling node.

Generating Fuzz Operator. This operator creates new random `cipher_suites` or `Extension` instances following the respective TPL formats and replaces the target subtree with the new instance. Even when the TPL format is followed by the generated subtree, the node bytes are mostly random.

These operators expect different kinds of data to alter and so they need to filter the available data to choose only what is appropriate to their function. The following filters were developed by Walz et al. for this purpose:

End nodes filter. Used by the *Fuzz Int* and *Fuzz Data* operators, this filter selects only the end nodes or leaves of the tree that point to specific bytes in the message.

Vector element filter. Used by the *Deleting* and *Voiding* operators, returns all items which are part of a Vector or list containing other items of the same type, e.g., compression methods, each of the included extensions, or the named curves in the `supported_groups` extension.

GeneratingFuzzOp filter. Used by the *Generating Fuzz* operator, selects only the elements which are either the `cipher_suites` list, the `extensions` list, or any of the `Extension` items in the list.

Dynamic length filter. Used by the remaining operators, returns all leaves and subtrees that have a corresponding field indicating their length, i.e., all the Vector and opaque fields.

By combining these components together, Walz et al. managed to generate huge amounts of mostly valid `ClientHello` messages for black-box TLS fuzzing. Due to the vast number of possible fields in protocol messages that trigger different execution variations, it is not enough to look for application crashes or memory corruption only. It is also important to monitor when the TLS server sends a `ServerHello` message, indicating that it thinks the sent data is valid and wishes to continue with the handshake, or when it sends an `Alert` message, indicating something went wrong during the communication. The approach by Walz et al. also takes note of the responses sent by the servers and writes them in a file for later manual analysis.

4.2.3 eGMT: Improvements on GMTs

We believed that the approach proposed by Walz and Sikora can be effective to test our target system as well. However, we found that their method can be improved. Even though, GMT works with `ClientHello` messages generated for TLS 1.3, it was initially developed to test TLS 1.2 and earlier versions. Therefore, GMT, and specifically the eTPL implementation, does not include structure definitions for new objects relevant to TLS 1.3 and thus, it does not mutate data in nodes specific for this version. Also, since this thesis focuses on TLS 1.3, their implementation contains (a lot of) irrelevant code.

In addition, we consider that the following points in the original tool could be improved:

- The *Voiding* and *Deleting* operators behave mostly in the same way. The only differences are the fact that the *Voiding* operator keeps a place marker while the *Deleting* operator does not, and that the data they are applied to is filtered differently. These operators could be merged into one.
- The *Fuzz Data* operator should not have a hardcoded option to preserve length. When this is the case, this operator is no different than the *Fuzz Int* operator in "full range mode".
- The *Appending Fuzz* operator could add valid (and not just random) data as a sibling of a specific node as well. For instance, if a new node will be appended as a sibling of a member of the `cipher_suites` or `signature_scheme_list` nodes, the new node could have a valid cipher suite or signature scheme taken from a corresponding list of values.
- Similarly, the *Generating Fuzz* operator could replace more objects other than `cipher_suites` and `Extensions` with semi-random data. It could be used, e.g., to replace the `protocol_version` or `handshake_type` for other valid values.
- While the original filters make sense for some of the operators, they tend to leave out nodes and subtrees where a specific operator could be applied. The *Duplicating* operator, for example, is applied to Vector Items only, but it could also be used to duplicate the bytes in an end node or a `key_exchange` field in a `key_share` extension. As another example, the *Voiding* operator is applied to dynamic length objects, which leaves out single `Extension` objects, length and type fields, where we think it could be useful.

We implemented a simpler version of Walz' and Sikora's GMT approach and added the aforementioned improvements and further extensions. Our implementation is denoted

extended GMT (eGMT) and it was added to our testing framework as an additional fuzzing tool. It focuses on TLS 1.3 and replaces the original complex data structures with easier to use Python dictionaries and lists for the subtrees and vectors. Just like for GMTs, the resulting objects are tree-like structures where each node has either bytes of data or a child subtree, as illustrated in Figure 11.

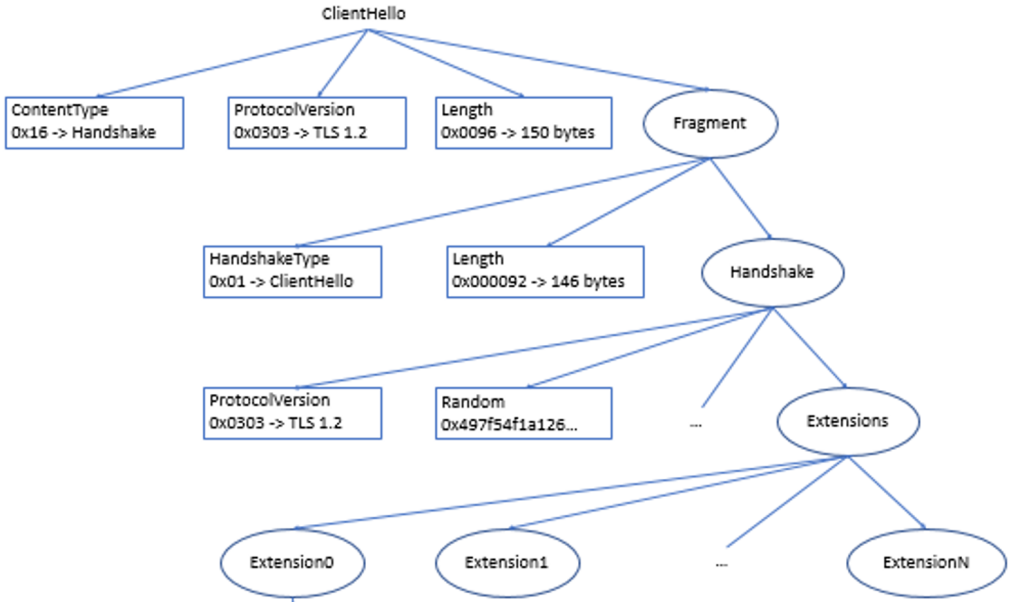

Figure 11: eGMT structure example for a `ClientHello` message. Circle nodes are parent nodes and square nodes are data or end nodes.

We started by coding an algorithm that behaved in the same way that the original implementation and that had the same operators. Then, we tweaked the operators and filters to meet our requirements, creating "enhanced" versions for each. The changes we made to the original operators are as follows:

eVoiding/Deleting Operator. Both original operators were merged into a single one that deletes all data in a node or subtree while keeping the place marker, which we consider is a useful feature. Also, we removed the filters and apply the final operator to any node or subtree in the GMT.

eDuplicating Operator. This operator was adjusted to work with any type of node or subtree and the filters were removed.

eTruncation Fuzz Operator. This was also adjusted to be applied globally. In the new implementation it can also truncate byte strings, such as the ones included in the `random` or `key_exchange` fields, or even whole subtrees, removing fields as necessary to get the new length once the tree is encoded back to bytes. This was not possible before.

eFuzz Int Operator. This operator was left unchanged.

eFuzz Data Operator. The length preservation option was set to be chosen at random which is probably what the authors intended originally. The probability of keeping the original byte length for a field is 1/2. Also, the filters were removed so this operator can now be used to replace all bytes in the `extensions` list, for example.

eAppending Fuzz Operator. This operator now has the capability to append valid data from a predefined list with a probability of 1/2 if the target node is a field in the list. Some of the fields that were added to the list are `protocol_version`, `handshake_type`, `cipher_suites`, `extension_type`, and `supported_groups`, to name a few. If the field is not found in the list, it simply generates a random byte stream of up to 4 bytes as before.

eGenerating Fuzz Operator. This operator now replaces any node or subtree in the GMT with semi-random data based on valid values and structures, if any. If the target node or subtree does not have a set of possible valid values, such as a length field, then, the operator behaves in the same way as the *Fuzz Data* operator.

The filters were removed for all but the *Fuzz Int* operator and the operators were changed to act differently based on the data they were applied to. Furthermore, the following new operators, not included in the original implementation, were added:

Zero Operator. Changes all the bytes in a node or subtree to zero bytes, maintaining the original byte length and structure. Uses the *End nodes* filter.

Bit Flip Operator. Selects up to five random bits from an end node (which contains raw bytes instead of other children or subtrees) and flips them. Uses the *End nodes* filter.

Rotation Operator. Works only on vector items. It swaps the place of the selected random operand with a sibling of the same type also selected at random. Uses the *Vector elements* filter.

To summarize, Table 4 shows the final operators in our eGMT fuzzer.

4.2.4 eGMTs on handshake messages

We further extended the work by Walz and Sikora by adding modules for key derivation, integrity checks, encryption and decryption of data, allowing us to fuzz other handshake messages. In these modules, we implemented features that were relevant for the target application (HTLS). Therefore, the encryption and decryption supports only AES in GCM mode, the signatures are for ECDSA, and the key exchange uses ECDHE. With these added features, we implemented modules for fuzzing `ServerHello`, `EncryptedExtensions`, `Certificate`, `CertificateVerify` and `Finished` messages, along with the original `ClientHello` message fuzzer.

The resulting algorithm is a black-box protocol fuzzer, but it is able to obtain the exit code for the given client or server application in order to detect crashes. Each fuzzing module has their own set of checks to decide whether the generated test case was accepted by the peer or not, as explained below. An accepted test case is written to a file and saved into an external directory for later inspection while its cryptographic hash is saved

Operator short name	Description
eVoidOp	Deletes all data in an operand but keeps a place marker
eDupplicatingOp	Duplicates the operand
eTruncationFuzzOp	Truncates the operand into a new random length
eFuzzIntOp	Sets the operand to a new random integer, using either proximity or full range mode
eFuzzDataOp	Sets the operand to new random bytes, keeping the original byte length with a probability of 1/2
eAppendingFuzzOp	Creates a new node of either up to 4 random bytes or valid data and adds it as a sibling of the operand
eGeneratingFuzzOp	Replaces any node or subtree with valid or random data
ZeroOp	Replaces all bytes in the operand with zero bytes
BitFlipOp	Flips up to five random bits from the operand
RotationOp	Swaps two vector items randomly

Table 4: Summary of final operators in eGMT

in memory to avoid writing the same files repeatedly. Test cases that trigger an exit code other than one or zero are considered crashes and are also written to a file and saved to disk, keeping their hashes in a separate list. This was added since the HTLS apps return an exit code of one when an expected failure or message rejection occurs, and zero when the application was executed successfully, i.e., the handshake was completed.

The algorithm performs the following steps depending on the handshake message to be fuzzed:

ClientHello

1. The fuzzer generates a new test case based on a valid message by applying a random number of mutations
2. The fuzzer connects to the server and sends the test case
3. If the server sent a response and the response has a `content_type` of type `handshake`, the test case is considered accepted. In any other case, the test case is considered rejected.

An accepted test case for a `ClientHello` message generally means that the fuzzer generated a mostly valid message that contains enough information for the server to continue the handshake without failing. Because of this, accepted cases need to be manually reviewed, since they do not always imply that a security bug is present. A `ClientHello` with additional garbage extensions, for example, or with extra appended bytes can be accepted by a TLS server and the RFC also allows it. If, however, a required extension or field is missing and the server still accepts the message, then there can be a security issue.

ServerHello

1. The fuzzer generates a new test case based on a valid message by applying a random number of mutations
2. The fuzzer listens for connections and waits for a client to connect
3. Once a client connects, the fuzzer receives the message from the client and stores it as a `ClientHello` message in the handshake context in memory, then, it sends the test case
4. If the client sends an `Alert` message, the test case is considered rejected
5. At this point, the fuzzer attempts to derive the cryptographic keys and continue the handshake with valid encrypted data
6. If after sending the `EncryptedExtensions` message the client sends an `Alert` of type `bad_record_mac`, then it is implied that the generated `ServerHello` was accepted but the MAC integrity check failed, and thus the derived keys were incorrect. Any other alert type means the test case was rejected
7. If the handshake continues to the end and the client also sends a `Finished` message, then the test case is not only considered accepted, but also it shows that the derived keys were correct. This is signaled with a small change in the file name when saving the accepted case to disk

For this `ServerHello` test it is frequently the case that the generated test case has a length larger than the amount of bytes that were actually sent, in which case, the client continues to expect data from the server before sending any kind of response. For this reason, it is important to continue through the whole handshake before deciding if a test case was accepted or not, since also in a valid handshake, the client will not send a response until the whole handshake is completed. As mentioned in step 7, this is also helpful to find test cases where the test case modifications still produced valid cryptographic keys for the session.

EncryptedExtensions

1. The fuzzer generates a new test case based on a valid message by applying a random number of mutations
2. The fuzzer listens for connections and waits for a client to connect
3. Once a client connects, the fuzzer receives the message from the client and stores it as a `ClientHello` message in the handshake context in memory, then, it sends a valid `ServerHello` message and derives the cryptographic keys from the handshake context
4. The generated test case is encrypted and sent to the client. If an `Alert` message is received, the test case is considered rejected
5. Again, the handshake continues while checking for each sent message if an `Alert` is received, in which case the test case was rejected
6. If the handshake is completed, the test case is considered accepted

For this message and the rest that will be explained next, there is no need to check whether the keys were derived correctly or not, since this happens before the actual fuzzing and it is assumed to have been done with all the required security parameters. For these messages, the generation of test cases is done before encryption so that once generated, the data can be properly encrypted and sent to the peer for processing.

For the `EncryptedExtensions` message, it is mostly important to check that the client complies with the RFC restrictions, e.g., invalid extensions should be ignored or extensions that were not requested in earlier messages should trigger an error.

Certificate

1. The fuzzer generates a new test case based on a valid message by applying a random number of mutations
2. The fuzzer listens for connections and waits for a client to connect
3. Once a client connects, the fuzzer receives the message from the client and stores it as a `ClientHello` message in the handshake context in memory, then, it sends a valid `ServerHello` message and derives the cryptographic keys from the handshake context
4. The fuzzer sends a valid `EncryptedExtensions` message with no additional extensions
5. Now the test case is generated for the `Certificate` message, which is encrypted and sent to the client. Just like before, an `Alert` at this point is considered a rejection
6. The next handshake messages are sent and if no `Alert` is received for any of them, the test case is considered accepted.

The basic fuzzing module for both the `EncryptedExtensions` and `Certificate` messages works in a similar way. We developed an additional extension to the `Certificate` message fuzzing that also considers fields in the ASN.1 structure for the X.509 certificate, which is discussed in the next subsection.

CertificateVerify

The algorithms for the `CertificateVerify` and `Finished` messages are slightly different than the previous ones, since both of these messages need to be generated using the transcript hash for the handshake, therefore, they are valid for each specific handshake. Then, these algorithms do not require a seed to initialize the test case generation, although an option exists for this in case there is a need to fuzz specific fields in the message regardless of the validity of the data. For both of these messages, the fuzzer creates the respective trees with valid data on-the-fly and performs the mutations on these generated trees, before sending to the peer.

1. The fuzzer listens for connections and waits for a client to connect
2. Once a client connects, the fuzzer receives the message from the client and stores it as a `ClientHello` message in the handshake context in memory, then, it sends a valid `ServerHello` message and derives the cryptographic keys from the handshake context

3. The fuzzer continues the handshake and generates the `CertificateVerify` message corresponding to the handshake after having sent the `Certificate` message

4. The fuzzer performs mutations on the generated message based on the selected parameters and sends it to the client. If an `Alert` is received, the test case was rejected

5. The fuzzer generates a valid `Finished` corresponding to the handshake and sends it to the client. If the client sends its own `Finished` message

Given that the `CertificateVerify` message includes an ECDSA signature, we included an extension of this algorithm that takes into account the ASN.1 objects while fuzzing, just like for the `Certificate` message.

Finished

1. If fuzzing from the server side, the fuzzer listens for connections and waits for a client to connect. Otherwise, the fuzzer connects to a listening server and sends an initial `ClientHello` message

2. When using server-side fuzzing, the fuzzer sends the rest of the handshake messages up to the `CertificateVerify` message, deriving the cryptographic keys and encrypting data when necessary

3. The fuzzer generates a valid `Finished` corresponding to the handshake and performs mutations on the generated message based on the selected parameters before sending to the peer

4. If fuzzing from the server side, receiving a `Finished` message from the client is considered a message acceptance, while anything else is a rejection. From the client side the check is slightly trickier, since the server normally expects additional messages from the client at this point. For this, the fuzzer sends an additional set of messages to check the state of the connection. If the connection remains open, the message is considered accepted, otherwise, it is considered rejected.

It is worth mentioning that specifically for the `CertificateVerify` and `Finished` messages there is a larger amount of false positives for accepted messages due to the fact that they are unique for each handshake, i.e., each test case has a different cryptographic hash. Due to the randomized nature of the mutations, there are cases were previous alterations are reverted (e.g., when fuzzing the bytes in a length field right before the fuzzer repairs all lengths in the object), which results in completely valid, unaltered data. For this reason, manually reviewing the reports is of great importance, distinguishing between relevant and non-relevant cases.

The algorithms discussed above are the steps followed by the fuzzer for each connection. In order to automate the fuzzing process, the fuzzer has the capability of executing the corresponding client or server instance as a child process, which also allows it to read the final exit code when the execution is terminated. We found during our tests that some mutated cases left an inconsistent state in the socket, making it unusable for several seconds even after killing the child process. To avoid unwanted latency and to make the process as smooth as possible, the fuzzer opens sockets in different available ports for each executed instance.

4.2.5 eGMTs on ASN.1

We also implemented our eGMT-based approach for fuzzing DER-encoded ASN.1 structures and test the parsing functions of the target application. This implementation first takes the bytes from a valid seed and decodes it into an eGMT structure made out of hierarchies of TLV fields. TLV Values in ASN.1 STRUCTURE types are other TLV objects and each field is a leaf in the tree. In this sense, these structures are different than the TPL objects in that every child always has a tag, length and value fields, where the value is either raw bytes or a child object. This concept is illustrated in Figure 12.

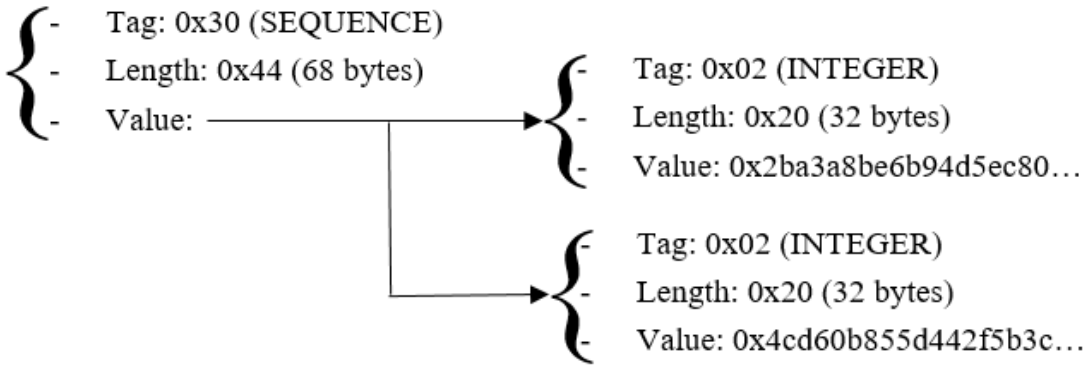

Figure 12: ASN.1 eGMT structure example for a ECDSA signature

eGMT ASN.1 objects are not as complex as TPL objects and thus, a simpler version of some operators was created for them. Initially, a custom version of each operator was developed for the ASN.1 structures, but as the code improved the ASN.1 fuzzer was adjusted to work with mostly the same operators as the TPL fuzzer. The only operators that had significant changes for the final version of the testing framework are the following:

ASN.1 Appending Fuzz Operator. Just like the final version in the TPL fuzzer, this operator generates new data to be appended as a sibling of the target node. For the ASN.1 fuzzer, however, it generates a whole new TLV object where the tag can be chosen with a probability of 1/2 to be a valid ASN.1 type from a list of most common types or a random byte. The length field is always chosen to be a random one-byte integer and the value field is filled with as many random bytes as indicated by the length.

ASN.1 Generating Fuzz Operator. This operator is much less complex than the one developed for the TPL fuzzer. It generates a new TLV object in the same fashion as the *ASN.1 Appending Fuzz* operator and replaces the current node or subtree with it.

The rest of the operators used for this implementation are the same as the final versions developed for the TPL fuzzer. The repairing mechanism and filters work in the same way, having all operators but the *eFuzzInt*, *Zero*, *BitFlip* and *Rotation* operator being applicable to any node or subtree in the test case (since both the *eFuzzInt* and *BitFlip* operators apply to end nodes, and the *Rotation* operator applies to vector items only). These four

operators also do not alter the length of the mutated node, so the repair method is skipped for them.

The algorithm was then used to generate test cases with seeds taken from valid ECDSA signatures, public keys and X.509 certificates, whose definitions were discussed in Section 2.1.10. The resulting data was used to fuzz the specific HTLS functions that used these structures by using binary applications developed specifically to use HTLS parsers and signature verification functions. This helped to find parsing bugs, e.g., where the application was accepting data with additional garbage bytes appended at the end of the stream, which is not allowed in DER encoding.

Algorithm 1 Randomized test case generation

function eGMT-Fuzz(V) ▷ Fuzz the tree V
  while $RND(\{true, false\})$ **do**
    $o \leftarrow$ $RND(\{O_{eVoid}, O_{eDup}, O_{Zero}, O^{eTrunc}_{fuzz}, O^{Int}_{fuzz}, O^{eData}_{fuzz}, O^{eApp}_{fuzz}, O^{eGen}_{fuzz}, O^{BitF}_{fuzz}, O^{Rot}_{fuzz}\})$ ▷ Select a random operator
    if $o \in \{O_{Zero}, O^{Int}_{fuzz}, O^{BitF}_{fuzz}, O^{Rot}_{fuzz}\}$ **then**
      $v \leftarrow RND(\{v \in V | F_o(v) = 1\})$ ▷ Select a suitable node v using filters
    else
      $v \leftarrow RND(\{v \in V\})$ ▷ Select any node v
    end if
    ApplyOperator$(()o, v)$ ▷ Apply operator o to node v
    if $v \notin \{O_{Zero}, O^{fuzz}_{Int}, O^{fuzz}_{BitF}, O^{fuzz}_{Rot}\}$ **then**
      Repair$(()v)$ ▷ Call the $REPAIR$ function
    end if
  end while
end function

Algorithm 2 Length repair function

function Repair(v)
  $fullRepair \leftarrow RND(\{true, false\})$
  while $v \neq \perp$ **do** ▷ Repeat until root is reached
    if $fullRepair$ **or** $RND(\{true, false\})$ **then**
      if IsAsn1Object(v) **then**
        RepairTlvLen(v) ▷ Repair DER length
      else
        RepairLocal(v) ▷ Repair TPL length
      end if
    end if
    $v \leftarrow$ ParentOf(v) ▷ Go up one level
  end while
end function

In addition, both TPL and ASN.1 fuzzing implementations were integrated to create additional fuzzing functions that take advantage of their functionality for target objects that contain both structures. This is particularly true for, e.g., `Certificate` messages (which can contain X.509 certificate and `SubjectPublicKeyInfo` structures) and `CertificateVerify`

messages (which contain ECDSA signatures). The resulting algorithm functions in the same way as the TPL fuzzer described in the previous section but now also takes into consideration each field in the ASN.1 structure, making the necessary changes when it detects that the target node belongs to an ASN.1 tree or a TPL tree. This also means, for instance, that the length fields in the ASN.1 tree can be repaired or the specific value fields can be fuzzed. However, the reparation of lengths in an ASN.1 structure needs to follow the DER encoding conventions.

Algorithms 1 and 2 show the final test case generation and length repair algorithms respectively, where RND is a function that returns an element from the given set.

These eGMT tests on ASN.1 show that eGMTs can be applied to other structured objects other than those specific for TLS 1.3. Potentially any protocol that sends bytes which can be arranged in specific hierarchical structures can be converted into eGMTs and fuzzed in the same way.

5 Results

This section describes all vulnerabilities and bugs that were found in the test environment after the static and fuzzing tests. HSSL promptly fixed all vulnerabilities when reported and an additional set of tests confirmed they were no longer present at the end of the assessment.

5.1 Summary

There were a total of 20 found security issues. These issues were divided in categories and had varying levels of severity and exploitability. The table in Figure 13 provides a summary of these issues, while the subsections that follow give a detailed explanation for each of them and their implications.

Issue	Application	Category	CVSS score	Found by
Aborted execution with wrong TLV length	HTLS	ASN.1 parsing	0.0	eGMT ASN.1 fuzzer
Unnecessary zero bytes in INTEGER values	HTLS	ASN.1 parsing	3.7	Wycheproof static cases
Missing prepended zero byte checks for negative int.	HTLS	ASN.1 parsing	0.0	eGMT ASN.1 fuzzer
BER style termination and length encoding	HTLS	ASN.1 parsing	3.7	Wycheproof static cases
NULL pointer dereference in Certificate parse	HTLS	ASN.1 parsing	3.1	eGMT fuzzer
Buffer over-read in log print with invalid certificates	HTLS	ASN.1 parsing	4.4	eGMT fuzzer
Garbage bytes after signature	HTLS	ECDSA bugs	3.1	eGMT ASN.1 fuzzer
Too short INTEGER values in signature	HTLS	ECDSA bugs	3.1	Wycheproof static cases
Missing ECDH public key validations	HTLS	ECDHE bugs	4.8	eGMT fuzzer
Errors in key formats	HTLS	ECDHE bugs	3.7	Wycheproof static cases
Buffer over-read in log print	HTLS	Memory errors	4.4/7.5	eGMT fuzzer
NULL pointer dereference in Finished message	HTLS	Memory errors	5.9	eGMT fuzzer
Non-zero compression method in ClientHello	HTLS	RFC deviation	3.7	eGMT fuzzer
More than one extension of the same type	HTLS	RFC deviation	0.0	eGMT fuzzer
Missing signature_algorithms extension	HTLS	RFC deviation	3.7	eGMT fuzzer
EncryptedExtensions can be sent unencrypted	HTLS	RFC deviation	0.0	eGMT fuzzer
Montgomery ladder error when x-coord. Is zero	Tinycrypt	ECDHE bugs	4.8	Wycheproof static cases
Infinity in the Montgomery ladder algorithm	Tinycrypt	ECDHE bugs	3.7	Wycheproof static cases
Edge case for Shamir multiplication in ECDSA	Tinycrypt	ECDSA bugs	5.9	Wycheproof static cases
Second edge case for Shamir multiplication in ECDSA	Tinycrypt	ECDSA bugs	5.9	Wycheproof static cases

Figure 13: Summary table of bugs found

The columns in the summary table refer respectively to the name of the issue, the library where it was found (either HTLS or Tinycrypt), the given category, the assigned CVSS score, and how it was found. In the *Found by* column the following tools are displayed:

- *Wycheproof static cases* refers to the lists of static test cases that are included in the Wycheproof project to test TLS implementations
- *eGMT ASN.1 fuzzer* refers to our eGMT-based implementation of ASN.1 fuzzing that is applied to binaries that perform specific functions for ASN.1 parsing, signature and certificate validations, and public key extractions without performing any other TLS actions
- *eGMT fuzzer* refers to our eGMT-based implementation of TPL fuzzing both with and without the integration with the eGMT ASN.1 fuzzer, which was applied to TLS handshake messages by using client and server binaries

5.2 CVSS

The Common Vulnerability Scoring System (CVSS) [115] score allows security researchers to estimate the exploitability and impact level of a given vulnerability through a numerical representation. The assigned scores help companies rank the threats in their infrastructure and prioritize them, allocating time and resources for remediation as convenient. CVSS has been adopted to measure the severity of published vulnerabilities by several reputable organizations and vulnerability databases, such as NVD by NIST [116], IBM [117], Oracle [118], CVE Details [119], etc.

However, CVSS also has some limitations and there are mixed opinions on how useful it really is. A vulnerability researcher at Carnegie Mellon University's CERT Coordination Center in the US published a blog post in 2015 pointing out that CVSS scores can be misleading without context and it fails to measure safety of human beings, especially when including new technologies, such as IoT [120]. More recently, in 2021, other researchers from the same institution criticized the numerical mappings to qualitative measurements in the CVSS formulae, stating they are not properly justified [121]. They also mention that CVSS fails to indicate the security risk of a vulnerability since it does not account for context, material consequences of a successful exploitation, and it involves operational problems, such as having vague guidelines which are understood differently by each security professional. This, they argue, makes it hard for companies to correctly identify which vulnerabilities should be tackled first.

Nevertheless, we considered that CVSS scores would be useful for our case to provide a point of reference that can help compare each of the found bugs with one another. Also, given the wide adoption of CVSS in the security community, these scores can be useful to measure the severity in comparison with other real-world examples, as those found in public vulnerability databases.

We used CVSS version 3.1 to measure these levels based on our own considerations for each bug. The CVSS system has three groups of metrics: the *Base* score assumes the impact of a vulnerability on a worst-case-scenario basis and takes into account its intrinsic characteristics, the *Temporal* score considers factors that can modify the score over time, such as the existence of simple to use exploits or official patches, and the *Environment* score considers changes to the previous scores when applied to a specific computing environment where there are, e.g., security controls that mitigate the impact. Given that the last two metric groups consider factors that are not entirely in scope for our target, such as the existence of exploit code or mitigations, we only include the Base scores.

As explained in the specification, the CVSS Base score, in turn, consists of two sets of metrics: the *Exploitability* set reflects how easy it is to exploit the vulnerability, and the *Impact* set reflects the consequences of exploiting the vulnerability [115].

5.2.1 Exploitability metrics

The Exploitability set includes the following metrics:

Attack Vector (AV). Refers to the context in which exploitation is possible. The longer the distance between the affected system in which the vulnerability can be exploited and the attacker, the higher the score. It can be one of:

- *Network (N).* Exploitation can happen from any point beyond the other options up to the entire Internet.

- *Adjacent Network (A).* Exploitation can happen from an adjacent network or topology at most.
- *Local (L).* Exploitation requires local system access or interaction from a user that has this access.
- *Physical (P).* Exploitation requires physical access to the vulnerable system.

Given the nature of the TLS protocol, the only value we considered for this metric in the vulnerabilities found is *Network (N).*

Attack Complexity (AC). Refers to the conditions that need to be met to make exploitation possible. The less complexity, the higher the score. This metric can have the following values:

- *Low (L).* There are no specialized conditions or extenuating circumstances and the vulnerability can be exploited with repeatable success.
- *High (H).* Exploitation requires conditions that cannot be controlled by the attacker or for which the attacker needs to prepare in advance (e.g., knowledge gathering, preparation of the target environment, or MitM preparation)

Privileges Required (PR). Refers to the privileges that the attacker needs in order to make exploitation possible. It can be one of:

- *None (N).* No special privileges are required. The attacker can exploit the vulnerability while being unauthorized.
- *Low (L).* The attacker needs user privileges that would normally allow access to data belonging to this specific user.
- *High (H).* The attacker needs administrative access across the vulnerable system.

User Interaction (UI). Refers to the need for the participation of a human user who is not the attacker to allow exploitation. Possible values are:

- *None (N).* No interaction from an additional user is required.
- *Required (R).* The attacker requires a user to take an action in the system for a successful exploitation.

5.2.2 Impact metrics

The Impact set includes the following metrics:

Confidentiality (C). Refers to the restriction of access to information for any unauthorized parties, allowing access only to authorized ones. Impact on confidentiality can be:

- *High (H).* Exploitation causes total confidentiality loss, allowing an attacker full access to the information that the system should protect or to a portion of information for which disclosure is a serious issue (e.g., an administrator's password).
- *Low (L).* Exploitation causes partial confidentiality loss, allowing an attacker limited information access or for which disclosure has no serious impact.
- *None (N).* Exploitation does not result in confidentiality loss.

Integrity (I). Refers to how trustworthy information can be in general. Impact on integrity can be:

- *High (H).* Exploitation causes total integrity loss, allowing an attacker to modify any protected files at will or files for which modification implies a serious impact. This applies not only to files but also to any information that is being transmitted in, e.g., network packets.
- *Low (L).* Exploitation allows some data modifications but these modifications have no serious impact or the attacker does not have control over the consequences of the modifications.
- *None (N).* Exploitation does not result in integrity loss.

Availability (A). Refers to the accessibility to the information resources provided by the affected system or component itself. Impact on availability can be:

- *High (H).* Exploitation causes total availability loss, allowing an attacker to fully deny access to information resources for new or existing users.
- *Low (L).* Exploitation causes service interruptions or reductions in performance, but the system is not completely unavailable.
- *None (N).* Exploitation does not result in availability loss.

5.2.3 Scope

The Base metric group includes, in addition to the Exploitability and the Impact sets, a **Scope (S)** metric that reflects whether exploitation of the vulnerability impacts resources other than the affected system or which are beyond the initial scope. This metric can have the following values:

- *Unchanged (U).* Exploitation affects only resources managed by the affected system or component, or by another impacted component that belongs to the same entity.
- *Changed (C).* Exploitation affects additional components belonging to different entities or which are beyond the scope of the original affected system.

5.2.4 Vector string and calculation

The CVSS score can be represented as a "vector string" composed of all the assigned metrics for a particular vulnerability preceded by a slash "/". Each metric is written in its abbreviated form followed by a colon ":" and then by the metric value. The vector string must also begin with the string "CVSS:" followed by the CVSS version that was used. Commonly, the order specified for the vector string (and the one used in this thesis) states the Exploitability metrics first, followed by the Scope metric and finally by the Impact metrics. For instance, a vulnerability that has the following assigned metrics: Attack Vector - Network, Attack Complexity - Low, Privileges Required - Low, User Interaction - None, Scope - Unchanged, Confidentiality impact - None, Integrity impact - Low, and Availability impact - None would be represented by the following vector string:

CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:N

The CVSS specification defines a set of equations that use each of these metrics to calculate the final numeric score. Each metric value has an associated number to be used in the equations, as shown in Table 5. The vulnerabilities discussed in the next section include both the vector string and the numerical CVSS score.

Metric	Metric Value	Numerical Value
Attack Vector	Network	0.85
	Adjacent	0.62
	Local	0.55
	Physical	0.2
Attack Complexity	Low	0.77
	High	0.44
Privileges Required	None	0.85
	Low	0.62 (or 0.68 if Scope is Changed)
	High	0.27 (or 0.5 if Scope is Changed)
User Interaction	None	0.85
	Required	0.62
Confidentiality / Integrity / Availability	High	0.56
	Low	0.22
	None	0

Table 5: Constant numeric values for CVSS metrics. Adapted from [115].

The equations for the Base metrics are the following:

$$
\begin{aligned}
ISS =& 1 - [(1 - Confidentiality) * (1 - Integrity) * (1 - Availability)] \\
Impact =& 6.42 * ISS \ \#\text{if Scope is Unchanged, or} \\
=& 7.52 * (ISS - 0.029) - 3.25 * (ISS - 0.02)^{15} \ \#\text{if Scope is Changed} \\
Exploitability =& 8.22 * AV * AC * PR * UI \\
BaseScore =& Roundup(Minimum[(Impact + Exploitability), 10]) \\
& \#\text{if Scope is Unchanged, or} \\
=& Roundup(Minimum[1.08 * (Impact + Exploitability), 10]) \\
& \#\text{if Scope is Changed}
\end{aligned}
$$

where ISS stands for Impact Sub-Score; AV, AC, PR, and UI refer to the numerical values for Attack Vector, Attack Complexity, Privileges Required, and User Interaction respectively; and *Roundup* and $Minimum$ represent functions. The $Minimum$ function, as the name implies, returns the lowest numerical value from the list of arguments. The *Roundup* function rounds the given number to the smallest that is equal to or smaller than the argument up to 1 decimal place, e.g., *Roundup*(5.02) returns 5.1 while *Roundup*(5.2) returns 5.2.

As an example, consider the CVSS score assigned to the vulnerability **NULL pointer dereference in Finished message** discussed in Section 5.3.4:

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:H - 5.9

Substituting the numerical values for each metric in the previous equations, we obtain:

$$
\begin{aligned}
ISS =& 1 - [(1 - Confidentiality) * (1 - Integrity) * (1 - Availability)] \\
=& 1 - [(1 - 0) * (1 - 0) * (1 - 0.56)] \\
=& 0.56 \\
Impact =& 6.42 * ISS \\
=& 6.42 * 0.56 \\
=& 3.595 \\
Exploitability =& 8.22 * AV * AC * PR * UI \\
=& 8.22 * 0.85 * 0.44 * 0.85 * 0.85 \\
=& 2.221 \\
BaseScore =& Roundup(Minimum[(Impact + Exploitability), 10]) \\
=& Roundup(Minimum[(3.595 + 2.221), 10]) \\
=& Roundup(Minimum[5.816, 10]) \\
=& Roundup(5.816) \\
=& 5.9
\end{aligned}
$$

5.3 HTLS Vulnerabilities

The following errors were discovered in the HTLS implementation only.

5.3.1 ASN.1 parsing

These are the issues that were found related to the ASN.1 parsing code. The issues were present in the decoding of ECDSA signatures and *SubjectPublicKeyInfo* (SPKI) objects. Sending data encoded with BER encoding instead of DER or with additional invalid values caused the program to fail, triggering buffer errors or unwanted behavior mainly in the ECDSA signature verification procedures.

Aborted execution with wrong TLV length. The ASN.1 fuzzer generated several test cases in which the length field of a TLV object in an ASN.1 structure was incorrect or missing. It was found that in these cases, the signature verification function in HTLS raised an error and aborted the execution, stopping the application completely without sending any response to the peer. This abort signal, however, was triggered by a failure of an `assert` condition, which is only present when compiling in *debug* mode. Productive applications are not normally compiled in debug mode and do not include `assert` statements, for which case this bug causes only a normal handshake failure and the affected peer does send an `Alert` message, causing no impact on security.

In a scenario in which the developer left debug asserts in production applications, this bug could have been used to cause denial of service, forcing a server shutdown by sending malicious requests, but given that the HTLS library is currently in development mode and it will be the responsibility of the developer using the library to compile production code correctly, this scenario is considered to be out of scope. HSSL fixed this issue by implementing buffer checks when reading TLV objects and calculating their remaining lengths.

Found after applying, e.g., the *BitFlipOperator* to one of the length fields in an ECDSA signature.

Score CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N - 0.0

Unnecessary zero bytes in INTEGER values. As mentioned in Section 2.1.10, BER-encoded (and consequently, DER-encoded) INTEGER values need to be prepended with a zero byte when the leftmost bit is set to "1" to avoid incorrect sign interpretations. The encoding rules also state that INTEGERs should have a byte length as short as possible, so prepended zero bytes should be avoided when not required. HTLS did not perform checks for INTEGER values larger than necessary and that included unnecessary zero bytes, thus allowing an indefinite amount of prepended zero bytes as long as the final integer was valid. It is recommended that these cases are not accepted when using DER encoding in order to limit the possibility of certain risks, such as, signature malleability attacks.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N - 3.7

Missing prepended zero byte checks for negative integers. After the previous bug was fixed, another bug was found in HTLS that caused the application to accept INTEGER values with the leftmost bit set and that did not include a prepended zero byte. As mentioned before, it is recommended that these values are rejected since it can lead to sign confusion and cause additional errors later. This was fixed by ensuring that zero bytes were indeed prepended when necessary, but no additional risks were found.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:N - 0.0

BER style termination and length encoding. As discussed in Section 2.1.10, the length of a value for a given TLV object is specified in the *length* field, where a single octet is used in the *short form* indicating a length of less than 127 bytes and two or more octets used in the *long form* indicate a larger length. HTLS was discarding INTEGER values with a length field that exceeded 4 octets (i.e., that had a length of more than 65'536 bytes), but for values lower than 4 octets but greater than 1 there were no checks implemented.

Furthermore, BER allows the use of the *long form* even when not necessary, e.g., when using a length encoding of "0x82 0x00 0x01" to indicate a length of a single byte instead of simply "0x01". This, however, is not allowed in DER and such values should be discarded. In addition, some test cases were using BER-style termination (two zero bytes "0x00 0x00"), which is used for indefinite length values not valid in DER.

HTLS was failing to check for these cases, allowing both BER-style length encoding and values of indefinite length. These cases are also recommended to be discarded to hinder the ability of an attacker to manipulate TLV objects, e.g., as in signature malleability attacks.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N - 3.7

NULL pointer dereference in certificate parse A NULL pointer dereference bug was found in the function responsible for parsing X.509 certificates. The bug happened when sending an invalid length value for the last BIT STRING in the certificate, which contains the signature required to validate the certificate chain. When reading the certificate, the parser first initializes a buffer structure to zero bytes. The structure for a X.509 certificate in HTLS looks similar to the following:

```
typedef struct {
    der_tlv certificate_tlv;
    der_tlv tbs_certificate_tlv;
    der_tlv signature_algorithm_tlv;
    der_tlv version_tlv;
    der_tlv serial_number_tlv;
    der_tlv signature_tlv;
    der_tlv issuer_tlv;
    der_tlv validity_tlv;
    der_tlv subject_tlv;
    der_tlv spki_tlv;
    der_tlv extensions_tlv;
    der_tlv signature_value_tlv;
    x509_parsed_exts extensions;
    unsigned char certificate_hash[32];
    size_t certificate_hash_len;
    unsigned char spki_hash[32];
    size_t spki_hash_len;
} cert_struct;
```

where `der_tlv` and `x509_parsed_exts` are other defined structure types. As the long list of components in the certificate is read, the structure is updated with pointers to each of the corresponding values. The problem was that for the variable that should point to the signature value (`signature_value_tlv`), if a length in the DER-encoded TLV was set

to a number greater than the amount of remaining bytes, the parser failed and the variable was left with a zero byte value, i.e., as a NULL pointer. The execution flow then continued and later when the application tried to access the pointer to the signature, it crashed.

The value of the affected variable cannot be directly manipulated and we did not find any way to set a value other than the valid signature pointer or a zero byte. Therefore, an attacker is not able to achieve any further exploitation with this bug other than a denial of service. Additionally, in a regular setup, a denial of service attack that abuses this bug affects clients only, i.e., the attacker needs to set a malicious server that sends a modified certificate (denying access to his own server) or set a MitM configuration and modify a `Certificate` message from a legitimate server, for which he would need the ECDH shared secret, so the impact is low. In a scenario where a server requires client certificates for authentication, this can indeed be used to crash the server and cause more damage.

Found with the eGMT TPL and ASN.1 integration after the fuzzer modified the length in the affected BIT STRING TLV before sending the `Certificate` message in the middle of a handshake.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:R/S:U/C:N/I:N/A:L - 3.1

Buffer over-read in log print with invalid certificates. A buffer over-read error was triggered from a function used to print verbose logs for debugging HTLS applications. This error occurs when a program is reading data from a buffer and, because of a bug, runs over the buffer boundaries, attempting to read from adjacent memory. This bug in particular occurred when parsing the SPKI value contained in an X.509 certificate. In this case, the length value of one of the first SEQUENCE objects in the certificate was incorrectly set to a value larger than the real one. This caused the parser to read the following objects at incorrect byte offsets, interpreting data bytes as other tag and length fields.

When HTLS tried to obtain the SPKI value, it first skipped the previous fields and set a pointer to where it calculated that the SPKI was. However, since the previous fields were wrong, the pointer to the SPKI was also wrong. HTLS would then use this pointer to calculate the length of the SPKI in memory by parsing an erroneous TLV length, which in most cases was a huge number, larger than the actual length of the whole certificate. Then, if the right logging level was set, HTLS attempted to read and print as many bytes from memory as indicated by the length variable, causing a huge memory dump which eventually crashed the application when it reached a restricted portion of memory. When the logging level was not set, the application simply failed to obtain the SPKI and the handshake failed, causing the application to exit safely.

In a scenario in which an attacker can access the contents of the logs (e.g., if she has local read access to a client or server machine), this vulnerability could allow her to dump and read large amounts of memory. If the dumped portions of memory contain sensitive information, such as private keys, the impact would increase significantly, in a similar way as for the Heartbleed vulnerability discussed in Section 2.3.2. This, however, adds complexity to the exploitation of the bug, since the attacker first needs this access and she also needs the logging level to be enabled for the application.

Found with the eGMT TPL and ASN.1 integration after the fuzzer modified the length in the affected SEQUENCE TLV, which caused the next objects to be invalid.

Score CVSS:3.1/AV:N/AC:H/PR:H/UI:N/S:U/C:H/I:N/A:N - 4.4

5.3.2 ECDSA bugs

Garbage bytes after signature. It was found that HTLS was not properly handling cases when extra "garbage" bytes were included in ECDSA signatures, as it was simply ignoring them after having parsed the relevant data. This causes the application to read and process more data than it needs and, even when the risk of this behavior is low, it is also recommended to reject such messages to avoid potential memory errors in the future.

Found after applying, e.g., the *eDuplicatingOperator* to the last object in a SPKI structure.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:R/S:U/C:N/I:N/A:L - 3.1

Too short INTEGER values in signature. An ECDSA signature contains at least two INTEGER values as shown in its ASN.1 format discussed earlier. These values are generally 32 bytes long, but this length is not a specific limit that applies to them in the encoding rules and they can vary.

In particular, there are cases when valid signatures have shorter integer values, as included in some of Wycheproof's test cases. HTLS was assuming a constant length of 32 bytes for both integers in a signature and, when doing the validation for these shorter cases, it still tried to read 32 bytes from memory. This caused the application to retrieve additional bytes of irrelevant data, causing the signature validation to fail when it should not have done so.

The impact of this bug is hard to measure. In theory, reading irrelevant bytes as part of the signature could cause a wrong signature to be validated correctly, allowing an attacker to fake possession of a private key and impersonate the server, for example. However, it is extremely hard for an attacker to meet the necessary conditions for this to happen, and he might need to make an indefinite amount of attempts before achieving a successful exploitation. If additional vulnerabilities are present, this could probably aid an attacker into manipulating the application in a malicious way.

Score (theoretic) CVSS:3.1/AV:N/AC:H/PR:N/UI:R/S:U/C:N/I:L/A:N - 3.1

5.3.3 ECDHE bugs

Missing ECDH public key validations. It was found that HTLS was not properly checking the public ECDHE key sent in the `key_exchange` extension in either the `ClientHello` or `ServerHello` messages. As explained in Section 2.1.5, an ECDH public key is a point on an elliptic curve and both parties need to verify that the public key sent by their peer does belong to the curve it is supposed to in order to avoid, e.g., invalid curve attacks [110]. In these attacks, researchers managed to obtain the private key corresponding to the server's public ECDH key. TLS 1.3, however, disallows non-ephemeral use of ECDH keys, making these attacks infeasible. In a scenario where the application caches ephemeral keys and reuse them in later connections (as done by some TLS implementations) the attacks could still be applicable. Still, it is unlikely that the keys are reused enough times to imply a high risk for this particular attack, since also the attacker needs to make several connections to the server hoping the server responds with the same key every time.

In addition to the missing public key validations, HTLS was also allowing any particular value to be used as a key, including a string of zero bytes. Attempting a key negotiation with a NULL key resulted in a NULL shared secret as well, from which other cryptographic

keys would be derived. An attacker could have abused this by changing the public keys sent in `ClientHello` and `ServerHello` messages via a MitM attack, allowing her to derive the negotiated keys and read and modify other handshake messages. However, full impersonation is not possible since the `CertificateVerify` message in the handshake cannot be forged by an attacker without access to the server's long-term private keys. Due to these controls inherently present in the TLS 1.3 protocol, we could not find any further exploitation that could be achieved from this bug, but it still was recommended to be remediated in HTLS in case any future vulnerabilities could bypass these controls.

Found after applying, e.g., the *ZeroOperator* to the `key_exchange` field in a `ClientHello` message.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:L/A:N - 4.8

Errors in key formats. The ECC implementation from Tinycrypt expects that private and public keys follow a specific format. This is explained in their repository documentation and it is up to the developers using the library to implement the necessary checks and changes to make sure the format is correct. It was found that HTLS was failing at checking that the keys complied with the following prerequisites:

1. Private keys should have a constant size of 32 bytes
2. Public keys should not include the uncompressed indicator (0x04)
3. Full-size private keys should not contain leading zeroes
4. If the private key has a length lower than 32 bytes, it should be padded with leading zeroes

These errors were causing unexpected behavior when calling functions from the library, such as reading from incorrect memory locations which changed bytes in the keys or generating incorrect secrets. The code was fixed to remove unnecessary leading zeroes, the uncompressed indicator in public keys, and to add leading zeroes for private keys that required it.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N - 3.7

5.3.4 Memory errors

The following are errors related to memory corruption or invalid memory accesses found in HTLS and that are not part of the previous categories. Even though these bugs were not found to be exploitable to the point of causing more severe vulnerabilities, such as a remote code injection, they could have brought future problems or caused unwanted behavior if left unfixed.

Buffer over-read in log print with empty messages. Another buffer over-read error was triggered from a function used to print verbose logs in both the client and server applications. The log function was found to be called in an incorrect manner when decoding the following handshake messages: `ClientHello`, `ServerHello`, `CertificateRequest`, `Certificate` and `CertificateVerify`.

The affected log print function walks through all the bytes in the received handshake message and, if the required log level is enabled, prints the hex representation of each byte to screen. This function needs as parameters a pointer to the start of the buffer that contains the message bytes and a number indicating the amount of bytes to be read from the buffer, it then tries to read as many bytes in the buffer as indicated by this amount.

The error occurred when calling the function from each of the handshake message respective decoders as the amount of bytes to be read was being calculated incorrectly. This amount was obtained with the subtraction

```
buf->end - buf->p
```

Where `buf` is the buffer containing all the bytes received, `buf->end` is a pointer to the end of the buffer, and `buf->p` is the pointer to the current read position, i.e., to the next byte that will be read. However, when an empty handshake message was sent (excluding the bytes in the Record header), the pointer to the end of the buffer was one position lower than the pointer to the current read position, so the subtraction resulted in a value of -1, which is represented in hex as 0xffffffff in 32-bit architectures and 0xffffffffffffffff in 64-bit.

Given that the byte amount sent as a parameter to the log print function is unsigned, the resulting value was read as a positive integer, translating to the number 4294967295 for 32-bit processors and 18446744073709551615 for 64-bit processors. The log print function would then attempt to read this amount of bytes, which would quickly turn into a read operation in a memory space that was restricted for the current process, triggering a segmentation fault error. When the required log level was not enabled, which is the most realistic scenario for any productive execution of a HTLS client or server, the log print function would not attempt to read restricted portions of memory but would still run through a seemingly endless loop until its internal counter reached such high numbers, causing the application to stall indefinitely.

Just like before, if a scenario exists where the required log level is enabled and the attacker has local read access for the application logs, she could trigger partial memory dumps and read sensitive data.

Score (scenario 1) CVSS:3.1/AV:N/AC:H/PR:H/UI:N/S:U/C:H/I:N/A:N - 4.4

On the other hand, since the application crashed immediately when logging was enabled or stalled indefinitely when not enabled, this could have been abused to cause denial of service attacks if left as is. The HSSL team fixed the issue by using a different variable that keeps track of the bytes left to read in the buffer, setting it as the amount of bytes that the log print function needs to read.

Score (scenario 2) CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H - 7.5

Found after applying, e.g., the *eVoidOperator* to the `handshake` field in a `ServerHello` message.

NULL pointer dereference in Finished message Similarly to the previous bug, it was found that sending an invalid `Finished` handshake message caused the application to crash with a memory error. This time, HTLS was trying a `memcpy` operation with two pointers in memory with a command similar to the following:

```
memcpy(mem->peer_verify_data, verify_data, 32);
```

Where `mem->peer_verify_data` is a pointer in a struct that keeps track of the data sent by the peer, and `verify_data` is a pointer variable containing the `Finished` hash from the buffer of received bytes. HTLS would later use this application to verify validity of the hash sent by the peer.

The `verify_data` variable is initialized to NULL and set before the `memcpy` operation by a function that is supposed to read 32 bytes of data from the memory buffer. However, when the buffer contains less than 32 bytes (due to, e.g., truncation or deletion), this function fails and the `verify_data` pointer remains as initialized, i.e., as a NULL pointer. This causes HTLS to attempt a `memcpy` operation using a NULL pointer as the source, which triggers a segmentation fault error and crashes the application.

We did not find a way to further exploit this bug since the *verify_data* cannot be directly modified, it can only be left as a NULL pointer. Also, since the `Finished` message is only sent and accepted at the end of a handshake, the attacker would need to complete a handshake and then send a correctly encrypted `Finished` message with a `Signature` field empty or shorter than 32 bytes. This is not unlikely in any normal client-server scenario since the attacker can pose as a normal client and craft a valid `Finished` message after completing a handshake. However, if the server requires a PSK or certificate for client authentication, the attacker would also need access to this information. In any case, just like before, an attacker can abuse this to cause denial of service, given that the affected application crashes immediately.

Found after applying, e.g., the *eTruncationOperator* to the `verify_data` field of a `Finished` message.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:H - 5.9

5.3.5 RFC deviations

The following bugs refer to situations in which the application did not behave as specified in the RFC [2]. These specifications in the RFC intend to mitigate some of the risks that were found for previous versions of TLS and all TLS 1.3 implementations should follow them. Not complying with the specification can lead to varying issues as discussed below.

Non-zero compression method in ClientHello Previous TLS versions supported compression methods, defined originally in RFC 3749 [122], that were meant to enable traffic compression before encryption. Compression methods had the goal of reducing the amount of bandwidth and latency involved in TLS sessions with large amounts of data. They, however, were found vulnerable to several kinds of attacks, such as the CRIME attack discussed in Section 2.3.4, after which it was decided that the best approach would be to remove them altogether.

TLS 1.3 does not support compression methods to avoid known vulnerabilities, but still includes the field in the `ClientHello` message for compatibility reasons, as mentioned in Section 2.2.3. In TLS 1.3, this field should be set to a single zero byte while anything different should trigger an error, but HTLS accepted `ClientHello` messages with non-zero bytes in this field. This could arguably be used to trigger a vulnerable server into using a compression method, if supported, facilitating attacks like CRIME on the session.

Found after applying, e.g., *eFuzzDataOperator* to the `legacy_compression_methods` field of a `ClientHello` message and then fixing lengths.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N - 3.7

More than one extension of the same type Extensions are subject to several restrictions in the RFC for TLS 1.3. One of these restrictions states that there must not be two or more extensions of the same type in a handshake message. It was found that HTLS accepted messages that did not comply with this restriction, allowing different extensions with the same type. This, however, was not found to result in any impact on security.

Found after applying, e.g., the *eDuplicatingOperator* to one of the extensions in a `ClientHello` message.

Score CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N - 0.0

Missing signature_algorithms extension The `signature_algorithms` extension is used to indicate which signature algorithms the server can use in `CertificateVerify` messages. The RFC states that all `ClientHello` messages must include this extension when using certificate authentication, otherwise, the server must abort the session. HTLS allowed `ClientHello` messages that did not include this extensions while authenticating the server with certificates. If allowed, this can potentially be abused to aid with downgrade attacks. However, such attacks can be extremely hard to achieve without the presence of additional vulnerabilities due to the differences between the handshakes of TLS 1.3 and lower versions, and given the protection mechanisms included in TLS 1.3 [3].

Found after applying, e.g., the *eVoidOperator* to the `signature_algorithms` extension of a `ClientHello` message.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N - 3.7

EncryptedExtensions can be sent unencrypted. It was found that the client application received and parsed an `EncryptedExtensions` message after a `ServerHello` message even when sent unencrypted. The RFC specifies that the `EncryptedExtensions` message must be sent encrypted since at this point both parties have enough information to derive the cryptographic keys. We did not find any attacks that could have abused this bug but it was still recommended to be fixed.

Found after applying, e.g., the *eAppendingOperator* which added extra bytes at the end of a valid `ServerHello` message. The bytes were found to be interpreted as an `EncryptedExtensions` message.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:N - 0.0

5.4 Tinycrypt vulnerabilities

While testing the HTLS application a set of vulnerabilities was discovered for the Tinycrypt library as well. These vulnerabilities were also fixed by the HSSL team and the fixed code was submitted to the original repository. The vulnerabilities that were found are described below.

[3]Servers that support TLS 1.3 but receive a negotiation for a lower version must set the last 8 bytes of their `random` value to a constant byte string which the client can check to detect downgrade attacks. Manipulation of the `random` value by an active attacker results in wrong transcript hashes, breaking the handshake altogether [2].

5.4.1 ECDHE bugs

Montgomery ladder error when x-coordinate is zero. Tinycrypt uses the Montgomery ladder with (X,Y)-only co-Z arithmetic for ECC scalar multiplication, discussed in Section 2.1.8. This requires that the Z coordinate is recovered at the end of the loop with the formula

$$Z = X_b y_P (x_P Y_b (X_1 - X_0))$$

It can be noted that if x_P is zero, then the value for Z will also be zero which is an error. Unfortunately, the curve NIST P-256 has two points where $x == 0$, namely

$$(0, 0x66485c780e2f83d72433bd5d84a06bb6541c2af31dae871728bf856a174f93f4)$$
$$(0, 0x99b7a386f1d07c29dbcc42a27b5f9449abe3d50de25178e8d7407a95e8b06c0b)$$

These two points are special cases for which the Z-recovery formula will not work. When attempting to compute a shared secret from these values, the corresponding HTLS function failed causing a NULL shared secret to be generated. This allows for a similar situation as explained before when the public keys are not properly verified, which an attacker can abuse to trick both parties into generating weak keys, although she would still need to bypass endpoint authentication to achieve hijacking of the session.

The fix implemented by HSSL consists of checking if the x-coordinate is zero at the beginning of the algorithm. If so, the z-coordinate is tracked after each addition via the formula $Z_{new} = Z_{old}(X_1 - X_0)$. Otherwise, the algorithm continues as normal. This helps to prevent errors caused by these cases and to generate proper cryptographic secrets.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:L/A:N - 4.8

Infinity in the Montgomery ladder algorithm. An edge case was found where at some point during the Montgomery ladder algorithm for scalar multiplication used in Tinycrypt, it was true that $R[0] = -R[1]$ and the next iteration required $R[0] + R[1]$, i.e., $P + (-P)$. This is exemplified in Figure 14. The sum would result in the point at infinity ∞, with which the Montgomery ladder algorithm does not work.

This bug was first detected by a vulnerability researcher in implementations by Oracle (CVE-2017-10176, fixed in [123]) and Mozilla (CVE-2017-7781, reported in [124]). It was found that often, libraries would implement the sum $P - P$ incorrectly, returning either $2 * P$ or throwing exceptions. This happened, for example, when the private key was close to the curve order. This issue is believed to allow partial manipulation of the ECDH secret, although an attacker would not have high chances of being able to determine the secret herself, making session hijacking unlikely. Even if the attacker had this capability, server impersonation would be infeasible without the private key corresponding to the authentication certificate sent later in the handshake.

The HSSL team fixed this issue by adding a check for whether any of the registers was the inverse of the other before making an addition that would result in the point at infinity. Then, the following iterations would be skipped for as long as the result would be the doubling of the point at infinity. When a different result can be obtained, the algorithm continues as normal.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N - 3.7

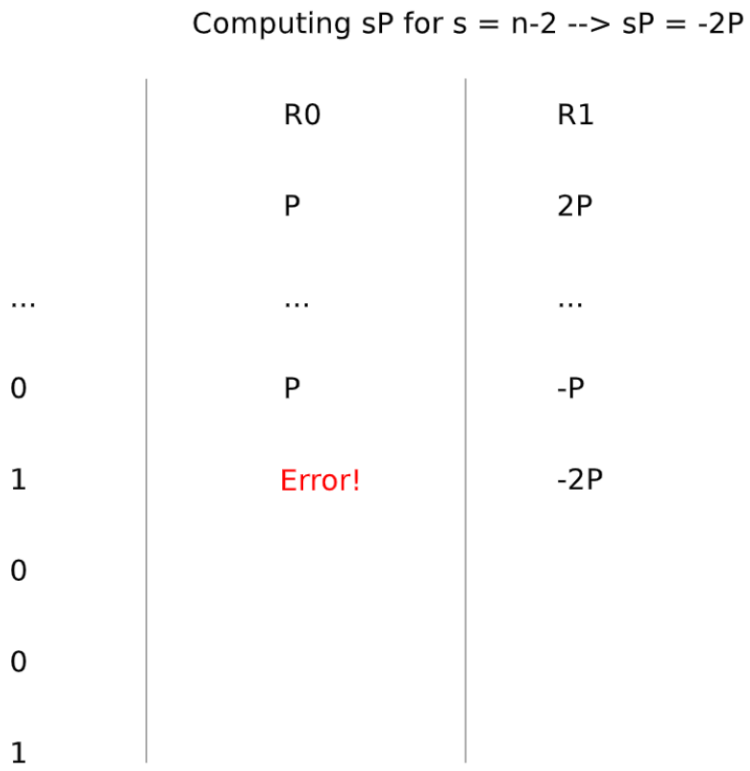

Figure 14: Infinity point error in Montgomery ladder

5.4.2 ECDSA bugs

Edge case for Shamir multiplication in ECDSA. As mentioned in Section 3.2, the ECC implementation used in Tinycrypt uses Shamir's trick (discussed in Section 2.1.9) for fast point multiplication when verifying ECDSA signatures. An edge case for this scenario can happen when at any point during the multiplication the value of S becomes either $-P$, $-G$, or $-(P+G)$. If this happens, the next addition in the algorithm can become ∞, which causes an error in the multiplication.

One way of solving this would be to rearrange the order of sums in the algorithm until an order is found where no addition results in ∞, however, these rearrangements can be computationally costly. Another solution, which was the one adopted for HTLS, is to implement a check that tells whether the next sum will result in ∞ and if so, then skip it, as it was done for the ECDH Montgomery ladder fix. This would mean that for that particular iteration, the point would not change, which is also valid in ECC since $P + \infty = P$.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:H/A:N - 5.9

Second edge case for Shamir multiplication in ECDSA. A second edge case in ECDSA signatures was found to cause errors when using Shamir's scalar multiplication. This case happens when for one iteration the coordinates for both points to be summed coincide, causing that $S = S + S$. Tinycrypt uses the Co-Z addition formulae for point addition regardless of whether the points to be summed are different or equal. Point

addition formulae work only when both points are different, but when the points are equal, point doubling should be used instead.

Similarly to the previous edge case, the chosen solution was to check in advance if the points to be summed are equal before actually performing the sum and use a point doubling instead of a point addition. HSSL submitted the fixes in the code for Tinycrypt by changing the algorithm as shown in Section 2.1.9 to the following:

```
point shamirs_trick(point G, point P, scalar u1, scalar u2) {
    point R[3], S;
    R[0] = 0; R[1] = G;
    R[2] = P; R[3] = P+G;
    int m1 = getNumBits(u1);
    int m2 = getNumBits(u2);
    int m = max(m1,m2);
    int i, b;
    S = P;
    for (i = m-2; i >= 0; i--) {
      b = (getBit(u1, i) << 1) | getBit(u2, i);
      if (x(S) == x(R[b]) {
        if (y(S) == -y(R[b])) {
          // S == -R[b], skip until non-0 add
        } else {
          // S == R[b], need to use doubling
          S = 2S;
        }
      else {
      S = S + R[b];
      }
    }
    return S;
}
```

Both of these edge cases cause errors that, if not properly fixed, could have led to incorrect validation of signatures or even to signature forgeries in the worst case scenario, i.e., allowing an attacker to create a signature that is always evaluated as valid.

Score CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:H/A:N - 5.9

6 Conclusions

In this thesis, we designed a testing framework, comprised of the eGMT fuzzer and the Wycheproof static test cases, for embedded implementations of the TLS 1.3 protocol. This framework was successfully used to find vulnerabilities in a development version of HTLS, a TLS implementation developed by Huawei targeted at embedded systems and constrained devices. This can also be used to test additional TLS implementations and measure their security. The testing framework was also part of the effort of making HTLS ready for open source release and the implemented tests will be included in the HTLS testsuite.

The testing framework was built using a combination of fuzzing methods and hand-crafted static test cases. For the fuzz testing, the main tool used was eGMT, an extended fuzzer based on previous research that generates new test cases by mutating tree-like structures obtained from valid handshake messages which are used as seeds. The generated test cases are then sent to a TLS client or server to fuzz the protocol at different stages of the TLS handshake, encrypting and decrypting messages as needed. The fuzzer monitors the responses sent by the target application as well as the exit code reported after execution to determine if a test case produced interesting behavior that could lead to software bugs. HSSL and I are preparing an article on eGMT.

Based on the obtained results, we believe that embedded implementations written in C have high probabilities of including software errors that can escalate to security vulnerabilities. Software implementations should be thoroughly tested before being released to productive applications. It is recommended to perform software tests (such as fuzzing) at different stages of the development process, especially after previously found bugs have been fixed to confirm that they are no longer present. In addition, developers should follow best coding practices to mitigate the risks of accidentally adding flaws in their applications and should receive security training when possible. Some examples of secure coding practices specific to C include implementing bounds checking when manipulating memory buffers, making sure that pointers are valid before use, and freeing previously allocated memory to avoid memory leaks.

The results obtained from this research indicate that our approach is a good direction towards extensive testing and uncovering of software errors in TLS libraries. However, there are aspects of our tool that should be addressed in future work, including:

- There is a lack of parallelization capabilities for the eGMT fuzzer. Higher fuzzing speeds could be achieved for the fuzzing process by adding, e.g., multi-threading features in processors that support it.

- There is inefficiency in generated test cases that have already been generated before. When a test case is repeated, it is still used and sent by the fuzzer to the client or server application, wasting precious time and computing power. Only after it results in an interesting behavior (e.g., it caused a crash or was accepted by the peer) it is checked to see if it was not generated before.

- The testing framework works correctly for the HTLS binary applications but, even when it would not be a hard task, adjusting it to be applicable to other TLS implementations would still require an additional coding effort. Ideally, the framework should be made as universal as possible such that an out-of-the-box execution is possible if needed.

- There will always be room for improvements in the mutation operators and fuzzing stages. Additional functionalities can be added for , e.g., longer lengths for randomly-generated strings, additional valid ASN.1 types and TPL structures (such as extensions), fuzzing of messages in later stages of a session (after a handshake has been completed), etc.
- This framework targets TLS 1.3, but it would be beneficial if previous versions can be tested with this same tool, providing a comprehensive solution for TLS applications that support more than one version. Including support for previous TLS versions, however, implies a substantial effort since it requires the inclusion of previously used handshake messages (such as `ClientCipherSpec` or `ClientKeyExchange`), cipher suites, and key derivation algorithms.
- Adding fuzzing capabilities for remote attestation implementations, such as attestation evidence verification code, would also be a good contribution and could provide interesting results.

References

[1] Statista, "Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030." https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/, 2022. Accessed on 2022-01-17.

[2] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Mozilla, 2018.

[3] P. Kotzias, A. Razaghpanah, J. Amann, K. G. Paterson, N. Vallina-Rodriguez, and J. Caballero, "Coming of Age: A Longitudinal Study of TLS Deployment," *Proceedings of the Internet Measurement Conference*, 2018.

[4] S. Kyatam, A. Alhayajneh, and T. Hayajneh, "Heartbleed Attacks Implementation and Vulnerability," in *2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, 2017.

[5] R. Seggelmann and M. Tüxen, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension," RFC 6520, Münster University of Applied Sciences, 2012.

[6] D. Munoz, "After All These Years, the World is Still Powered by C Programming." https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming. Accessed on 2022-06-02.

[7] "The Linux Kernel documentation." https://www.kernel.org/doc/html/v4.14/index.html#, 2016. Accessed on 2022-06-02.

[8] R. C. Seacord, *Secure Coding in C and C++, Second Edition*. Pearson, 2013.

[9] F. Pfenning, "Lectures Notes on Type Safety," *15-312: Foundations of Programming Languages*, 2004.

[10] J. Pan, "Software Testing," *18-849b Dependable Embedded Systems*, 1999.

[11] H. Liang, X. Pei, X. Jia, and J. Zhang, "Fuzzing: State of the Art," in *IEEE Transactions on Reliability*, vol. 67, pp. 1199–1218, 2018.

[12] J. Somorovsky, "Systematic Fuzzing and Testing of TLS Libraries," *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1492–1504, 2016.

[13] A. Walz and A. Sikora, "Exploiting Dissent: Towards Fuzzing-Based Differential Black-Box Testing of TLS Implementations," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, pp. 278–291, 2020.

[14] H. Böck, J. Somorovsky, and C. Young, "Return Of Bleichenbacher's Oracle Threat (ROBOT)," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 817–849, USENIX Association, 2018.

[15] V. A. B. Pop, A. Niemi, V. Manea, A. Rusanen, and J.-E. Ekberg, "Towards Securely Migrating Webassembly Enclaves," in *Proceedings of the 15th European Workshop on Systems Security*, EuroSec '22, (New York, NY, USA), pp. 43–49, Association for Computing Machinery, 2022.

[16] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "PACStack: an authenticated call stack," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 357–374, USENIX Association, 2021.

[17] Google, "Project Wycheproof." https://github.com/google/wycheproof. Accessed on 2022-04-01.

[18] Intel, "Tinycrypt." https://github.com/intel/tinycrypt. Accessed on 2022-03-02.

[19] E. Barker, A. Roginsky, and R. Davis, "NIST SP 800-133 Revision 2. Recommendation for Cryptographic Key Generation," standard, National Institute of Standards and Technology, 2020.

[20] A. Niemi, V. Bogdan, and J.-E. Ekberg, "Trusted Sockets Layer: A TLS 1.3 Based Trusted Channel Protocol," in *Secure IT Systems, 26th Nordic Conference, NordSec 2021*, pp. 175–191, 2021.

[21] D. Dolev and A. C. Yao, "On the Security of Public Key Protocols," in *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, pp. 350–357, 1981.

[22] "Secure Hash Standard (SHS)," standard, National Institute of Standards and Technology, 2015.

[23] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2015.

[24] NIST, "NIST FIPS 202. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," standard, 2015.

[25] "Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family." https://csrc.nist.gov/news/2007/request-for-candidate-algorithm-nominations, 2007. Accessed on 2022-05-21.

[26] X. Wang, Y. L. Yin, and H. Yu, "Finding Collisions in the Full SHA-1," in *CRYPTO, Lecture Notes in Computer Science*, vol. 3621, pp. 17–36, Springer, 2005.

[27] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full sha-1," in *Advances in Cryptology – CRYPTO 2017* (J. Katz and H. Shacham, eds.), (Cham), pp. 570–596, Springer International Publishing, 2017.

[28] NIST, "NIST FIPS 197. Advanced Encryption Standard (AES)," standard, 2001.

[29] "Announcing Development of a Federal Information Processing Standard for Advanced Encryption Standard." https://csrc.nist.gov/news/1997/announcing-development-of-fips-for-advanced-encryp, 1997. Accessed on 2022-05-21.

[30] W. Diffie and M. E. Hellman, "New Directions in Cryptography," in *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, 1976.

[31] B. Schneier, *Applied Cryptography. Protocols, Algorithms, and Source Code in C*. Wiley, 1996.

[32] J. H. Ellis, "CESG Research Report No. 3006: THE POSSIBILITY OF SECURE NON-SECRET DIGITAL ENCRYPTION," 1970.

[33] S. Levy, "The Open Secret." https://www.wired.com/1999/04/crypto/, 1999. Accessed on 2022-06-24.

[34] R. Barbulescu, P. Gaudry, A. Joux, and E. Thomé, "A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic," in *Advances in Cryptology – EUROCRYPT 2014* (P. Q. Nguyen and E. Oswald, eds.), (Berlin, Heidelberg), pp. 1–16, Springer Berlin Heidelberg, 2014.

[35] I. F. Blake, X. Gao, R. C. Mullin, S. A. Vanstone, and T. Yaghoobian, "The Discrete Logarithm Problem," in *Applications of Finite Fields*, pp. 115–138, Springer US, 1993.

[36] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Möller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)," RFC 4492, SafeNet and Sun Microsystems and Sun Labs and Corriente and Ruhr-Uni Bochum, 2006.

[37] C. Paar and J. Pelzi, *Understanding Cryptography*. Springer, 2010.

[38] D. J. Bernstein and T. Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography." https://safecurves.cr.yp.to/. Accessed on 2022-06-30.

[39] D. Brown, "Standards for Efficient Cryptography. SEC 1: Elliptic Curve Cryptography," standard, Certicom Research, 2009.

[40] D. Brown, "Standards for Efficient Cryptography. SEC 2: Recommended Elliptic Curve Domain Parameters," standard, Certicom Research, 2010.

[41] E. Barker, L. Chen, A. Roginsky, A. Vassilev, and R. Davis, "NIST SP 800-56A Revision 3. Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography," standard, National Institute of Standards and Technology, 2018.

[42] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)," standard, 2005.

[43] "Digital Signature Standard (DSS)," standard, National Institute of Standards and Technology, US, 2013.

[44] M. Rivain, "Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves.," in *IACR Cryptology ePrint Archive*, vol. 2011, p. 338, 2011.

[45] R. Goundar, M. Joye, and A. Miyaji, "Co-Z Addition Formulæ and Binary Ladders on Elliptic Curves," in *Cryptographic Hardware and Embedded Systems, 12th International Workshop – CHES 2010 of Lecture Notes in Computer Science*, vol. 6225, pp. 65–79, Springer, 2010.

[46] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Berlin, Heidelberg: Springer-Verlag, 2003.

[47] N. Méloni, "New Point Addition Formulae for ECC Applications," in *WAIFI: Workshop on the Arithmetic of Finite Fields*, (Madrid, Spain), pp. 189–201, 2007.

[48] A. Venelli and F. Dassance, "Faster side-channel resistant elliptic curve scalar multiplication," in *Arithmetic, Geometry, Cryptography and Coding Theory 2009*, vol. 521, pp. 29–40, American Mathematical Society, 2010.

[49] B. S. Kaliski, "A Layman's Guide to a Subset of ASN.1, BER, and DER." http://luca.ntop.org/Teaching/Appunti/asn1.html, 1993. Accessed on 2022-01-22.

[50] ITU-T, "Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation," standard, 2021.

[51] E. Wouters, E. Tschofenig, J. Gilmore, S. Weiler, and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," RFC 7250, Red Hat, ARM Ltd., Electronic Frontier Foundation, Parsons and INSIDE Secure, 2014.

[52] 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, NIST, Microsoft, Trinity College Dublin, Entrust and Vigil Security, 2008.

[53] ITU-T, "Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)," standard, 2021.

[54] I. Ristic, *Bulletproof SSL and TLS*. Feisty Duck, 2016.

[55] A. Freier, P. Karlton, and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0," RFC 6101, Netscape Communications, 2011.

[56] E. Rescorla, *SSL and TLS Designing and Building Secure Systems*. Addison-Wesley, 2001.

[57] R. Opplinger, *SSL and TLS Theory and Practice, Second Edition*. Artech House, 2016.

[58] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, Certicom, 1999.

[59] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, RTFM, Inc., 2006.

[60] S. Blake-Wilson and M. Nystrom, "Transport Layer Security (TLS) Extensions," RFC 4366, RSA Security, 2006.

[61] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, RTFM, Inc., 2008.

[62] D. Eastlake, J. Schiller, and S. Crocker, "Randomness Requirements for Security," RFC 4086, Motorola Laboratories and MIT, 2005.

[63] C. Calude, "Quantum Randomness: From Practice to Theory and Back," in *The Incomputable: Journeys Beyond the Turing Barrier*, pp. 169–181, 2017.

[64] NIST, "NIST SP 800-90A Revision 1. Recommendation for Random Number Generation Using Deterministic Random Bit Generators," standard, 2015.

[65] I. Goldberg and D. Wagner, "Randomness and the Netscape Browser," 1996.

[66] Debian, "DSA-1571-1 openssl – predictable random number generator." https://www.debian.org/security/2008/dsa-1571. Accessed on 2022-02-22.

[67] L. Dorrendorf, Z. Gutterman, and B. Pinkas, "Cryptanalysis of the Random Number Generator of the Windows Operating System," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security*, pp. 476–485, 2007.

[68] NetBSD, "RNG Bug May Result in Weak Cryptographic Keys (REVISED)." https://ftp.netbsd.org/pub/NetBSD/security/advisories/NetBSD-SA2013-003.txt.asc. Accessed on 2022-02-22.

[69] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of widespread weak keys in network devices," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[70] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security," RFC 4347, RTFM, Inc. and Stanford University, 2006.

[71] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347, RTFM, Inc. and Google, Inc., 2012.

[72] I. Ghafoor, I. Jattalai, S. Durranit, and M. T. Ch, "Analysis of OpenSSL Heartbleed Vulnerability for Embedded Systems," in *17th IEEE International Multi Topic Conference*, 2014.

[73] D. Bleichenbacher, "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1," in *Advances in Cryptology — CRYPTO '98*, pp. 1–12, 1998.

[74] T. Jager, J. Schwenk, and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[75] D. Fisher, "CRIME Attack Uses Compression Ratio of TLS Requests as Side Channel to Hijack Secure Sessions." https://threatpost.com/crime-attack-uses-compression-ratio-tls-requests-side-channel-hijack-secure-sessions-091312/77006/, 2012. Accessed on 2022-05-19.

[76] T. Duong and J. Rizzo, "Here Come The XOR Ninjas," 2011.

[77] J. W. Duran and S. Ntafos, "A Report on Random Testing," in *Icse '81. Proceedings of the ACM SIGSOFT International Conference on Software Engineering (ICSE'81)*, pp. 179–183, 1981.

[78] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, p. 32–44, 1990.

[79] V. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," in *IEEE Transactions on Software Engineering*, vol. 47, pp. 2312–2331, 2021.

[80] B. Byfield, "Fuzzing: An Old Testing Technique Comes of Age." https://thenewstack.io/fuzzing-old-testing-technique-comes-age/, 2017. Accessed on 2022-06-27.

[81] Google, "Honggfuzz. Security Oriented Software Fuzzer." https://github.com/google/honggfuzz. Accessed on 2022-02-03.

[82] M. Zalewski, "American Fuzzy Lop - A Security-Oriented Fuzzer." https://github.com/google/AFL. Accessed on 2022-02-03.

[83] Google, "OSS-Fuzz - Continuous Fuzzing for Open Source Software." https://github.com/google/oss-fuzz. Accessed on 2022-02-03.

[84] D. A. Wheeler, "How to Prevent the next Heartbleed." https://dwheeler.com/essays/heartbleed.html, 2014. Accessed on 2022-06-27.

[85] H. Böck, "How Heartbleed could've been found." https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html, 2015. Accessed on 2022-06-27.

[86] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[87] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, vol. 40, pp. 190–200, 2005.

[88] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A Taint Based Approach for Smart Fuzzing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 818–825, 2012.

[89] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proceedings of the USENIX conference on Annual Technical Conference*, pp. 309–318, 2012.

[90] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of C uninitialized memory use in C++," in *Proceedings of the 13th Annual IEEE/ACM International Symposium Code Generation and Optimization*, pp. 46–55, 2015.

[91] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Z. Fern, E. Eide, and J. Regehr, "Taming Compiler Fuzzers," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 197–208, 2013.

[92] K. A. Saleh, *Software Engineering*. J Ross Publishing, 2009.

[93] MozillaSecurity, "Funfuzz." https://github.com/MozillaSecurity/funfuzz. Accessed on 2022-06-30.

[94] X. Mendez, "wfuzz: Web application fuzzer." https://github.com/xmendez/wfuzz. Accessed on 2022-06-30.

[95] D. Jones, "Trinity: Linux system call fuzzer." https://github.com/kernelslacker/trinity. Accessed on 2022-06-30.

[96] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," in *Proceedings of the Network and Distributed System Security Symposium*, pp. 151–166, 2008.

[97] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[98] P. Godefroid, M. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *ACM Queue*, vol. 10, p. 20, 2012.

[99] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. X. Song, "Input generation via decomposition and re-stitching: finding bugs in Malware," in *Proceedings of the 2010 ACM Conference on Computer and Communications Security*, pp. 413–425, 2010.

[100] U. Kargén and N. Shahmehri, "Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 782–792, 2015.

[101] "LibFuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html. Accessed on 2022-06-30.

[102] J. Forrester and B. Miller, "An empirical study of the robustness of Windows NT applications using random testing," *in Proceedings of the 4th conference on USENIX Windows Systems Symposium*, vol. 4, p. 6, 2000.

[103] MozillaSecurity, "Peach." https://github.com/MozillaSecurity/peach. Accessed on 2022-06-30.

[104] V. A. B. Pop, "Secure migration of WebAssembly-based mobile agents between secure enclaves," Master's thesis, University of Turku, 2021.

[105] K. MacKay, "micro-ecc." https://github.com/kmackay/micro-ecc. Accessed on 2022-03-20.

[106] V. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: A Greybox Fuzzer for Network Protocols," in *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, 2020.

[107] "BooFuzz: A fork and successor of the Sulley Fuzzing Framework." https://github.com/jtpereyda/boofuzz. Accessed on 2022-05-16.

[108] S. Vaudenay, "Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS...," in *Advances in Cryptology – EUROCRYPT 2002*, vol. 2332, Springer, 2002.

[109] B. Möller, T. Duong, and K. Kotowicz, "This POODLE Bites: Exploiting The SSL 3.0 Fallback," 2014.

[110] T. Jager, J. Schwenk, and J. Somorovsky, "Practical Invalid Curve Attacks on TLS-ECDH," in *Proceedings of the 20th European Symposium on Research in Computer Security*, pp. 407–425, 2015.

[111] J. de Ruiter and E. Poll, "Protocol State Fuzzing of TLS Implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 193–206, USENIX Association, 2015.

[112] 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*, pp. 535–552, 2015.

[113] M. Kikuchi, "CCS Injection Vulnerability." http://ccsinjection.lepidum.co.jp/, 2014. Accessed on 2022-06-24.

[114] A. Walz and A. Sikora, "eTPL: An Enhanced Version of the TLS Presentation Language Suitable for Automated Parser Generation," in *Proceedings of the 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, pp. 810–814, 2017.

[115] I. FIRST, "Common Vulnerability Scoring System version 3.1 Specification Document Revision 1," specification.

[116] NIST, "National Vulnerability Database." https://nvd.nist.gov/. Accessed on 2022-06-30.

[117] IBM, "IBM X-Force Exchange." https://exchange.xforce.ibmcloud.com/. Accessed on 2022-06-30.

[118] Oracle, "Critical Patch Updates, Security Alerts and Bulletins." https://www.oracle.com/security-alerts/. Accessed on 2022-06-30.

[119] C. Details, "CVE Details - The ultimate security vulnerability datasource." https://www.cvedetails.com/. Accessed on 2022-06-30.

[120] D. Klinedinst, "CVSS and the Internet of Things." Carnegie Mellon University's Software Engineering Institute Blog, 2015. Accessed on 2022-06-29.

[121] J. M. Spring, E. Hatleback, A. D. Householder, A. Manion, and D. Shick, "Time to Change the CVSS?," in *IEEE Security and Privacy*, vol. 19, pp. 74–78, IEEE.

[122] S. Hollenbeck, "Transport Layer Security Protocol Compression Methods," RFC 3749, VeriSign, Inc., 2004.

[123] Oracle, "Oracle Critical Patch Update Advisory - July 2017." https://www.oracle.com/security-alerts/cpujul2017.html. Accessed on 2022-05-06.

[124] Mozilla, "Issue with elliptic curve addition in mixed Jacobian-affine coordinates." https://bugzilla.mozilla.org/show_bug.cgi?id=1352039. Accessed on 2022-05-06.