Institute for Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Master Thesis

Symbolic-Model-Guided Fuzzing of Cryptographic Protocols

Maximilian Ammann
Matrikel-Nr.: 1471541

Time Period:	28. April bis 28. October 2021
Primary Reviewer:	Prof. Dr. Alexander Knapp
Secondary Reviewer:	Prof. Dr. Gidon Ernst
Supervisors:	Dr. Lucca Hirschi & Dr. Steve Kremer

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, der 18. Oktober 2021 Maximilian Ammann

Abstract

Fuzzing implementations of cryptographic protocols is challenging. In contrast to traditional fuzzing of file formats, cryptographic protocols require a specific flow of cryptographic and mutually dependent messages to reach deep protocol states. The specification of the Transport Layer Security (TLS) protocol describes sound flows of messages and cryptographic operations. Although the specification has been formally verified multiple times with significant results, a gap has emerged from the fact that implementations of the same protocol have not undergone the same logical analysis. Because the development of cryptographic protocols is error-prone, multiple security vulnerabilities have already been discovered in implementations in TLS which are not present in its specification.

Inspired by symbolic protocol verification, we present a reference implementation of a fuzzer named TLS Protocol Under FuzzINg (tlspuffin) which employs a concrete semantic to execute TLS 1.2 and 1.3 symbolic traces. In fact attacks which mix TLS versions are in scope of this implementation. This method allows us to utilize a genetic fuzzing algorithm to fuzz protocol flows, which is described by the following three stages. 1) By mutating traces we can deviate from the specification to test logical flaws. 2) Selection of interesting protocol flows advance the fuzzing procedure. 3) A security violation oracle supervises executions for the absence of vulnerabilities.

The novel approach allows rediscovering known vulnerabilities, which are out-of-scope for classical bit-level fuzzers. This proves that it is capable of reaching critical protocol states. In contrast to the promising methodology no new vulnerabilities were found by tlspuffin. This can can be explained by the fact that the implementation effort of TLS protocol primitives and extensions is high and not all features of the specification have been implemented. Nonetheless, the innovating approach is promising in terms of quickly reaching high edge coverage, expressiveness of executable protocol traces and stable and extensible implementation.

Acknowledgement

The past six months were an extraordinary experience. While working on this master thesis I had the chance to follow my curiosity, experienced great hospitality and worked with the most amazing people.

Foremost, I would like to express my deepest appreciation for my supervisors Lucca Hirschi and Steve Kremer, who work at LORIA in Nancy. During the past six months, we had plenty of discussions and discovered demanding challenges. Even though, the pandemic situation put up enormous challenges in the beginning, my supervisors put great effort in providing a space for discussions and learning, even if that meant to move the office into the garden and work on a children's whiteboard[1]. Steve and Lucca supported me not only with lessons about formal methods, but also in every way I could imagine. For example, when I was required to do a COVID test in Nancy, and had no clue how to get one without a Card Vitale. I cannot begin to express my thanks to Lucca and Steve for reviewing this thesis extensively.

Moreover, I would like to extend my sincere thanks to Alexander Knapp who provided me with reviews and advise during the writing of this thesis. I am also grateful for the courses on formal methods by Alex during the past few years during my master. Without the knowledge and feel for formal methods I acquired there, I would have missed essential skills.

I would also like to extend my deepest gratitude to my friends in Nancy. Their hospitality allowed me to find times for relaxation. We did great trips to the Vosges and nearby lakes. Apart from that we regularly went bouldering, had beers afterwards and talked about our scientific work. Without my friends there, the time during a global pandemic would have been way more challenging.

Special thanks to the authors of the LibAFL and the fuzzing community, who supported me with bug fixes in upstream libraries or helped me to debug issues. They showed me, that fuzzing is a fascinating topic and that many people are actively researching in this area.

Last, but not least I would like to thank my family and friends here in Germany. I would like to thank my loved one for listening to my final presentation several times. Moreover, I also wish to thank my friends from numerous voluntary projects, who supported me by taking over work and therefore relieving me during the last half year.

[1]`https://twitter.com/MarieKremer3/status/1387395493307850755`

Contents

1 Introduction **1**
1.1 Contributions . . . 2
1.2 Outline . . . 3

2 Foundations of the Transport Layer Security Protocol **5**
2.1 Handshake Overview . . . 6
2.2 Key Schedule . . . 8
2.3 Session Resumption . . . 10
2.4 Security Properties and Known Attacks . . . 11

3 Foundations of Fuzzing **13**
3.1 Terminology . . . 13
3.2 Genetic Fuzzing Algorithm . . . 14
3.3 Dimensions of Fuzzing . . . 15
3.4 Related Work and State-of-the-Art Protocol Fuzzing . . . 16

4 Term Rewriting and Symbolic Traces **19**
4.1 Term Algebra . . . 19
4.2 Equational Theories and Term Rewriting . . . 21
4.3 Symbolic Traces . . . 24
4.4 Semantics of Symbolic Traces . . . 25

5 Designing a Symbolic-Model-Guided Fuzzer **29**
5.1 Fuzzing Harness . . . 30
5.1.1 Agents . . . 30
5.1.2 Traces as Input Space . . . 31
5.1.3 Auto Extraction of Knowledge . . . 33
5.1.4 Knowledge Queries . . . 34
5.2 Security Violation Oracle . . . 36
5.3 Fuzzing Algorithm . . . 40
5.3.1 Multi-Processing Architecture . . . 42
5.3.2 Seeds . . . 43
5.3.3 Mutations . . . 44
5.4 Optimizations . . . 49
5.4.1 Errors and Robustness . . . 49
5.4.2 Recipe Size Explosion . . . 50

6 Implementing a Symbolic-Model-Guided Fuzzer **53**
6.1 Ecosystem and Programming Language . . . 54
6.2 Library Under Test . . . 54
6.3 Trace Overview . . . 56

6.4 Term Implementation . . . 58
6.4.1 Term Syntax . . . 58
6.4.2 Knowledge Query Syntax . . . 60
6.4.3 Term Data Structure . . . 61
6.4.4 Term Visualization . . . 63
6.5 Test Input Serialization . . . 64
6.6 Functional TLS Client . . . 65
6.7 Instrumentation . . . 67
6.7.1 Code Coverage Instrumentation . . . 67
6.7.2 Security Violation Oracle Instrumentation . . . 68
6.7.3 Memory Sanitization . . . 69
6.8 Fuzzer Statistics . . . 70
6.9 Development Methodology . . . 71
6.10 Command-line Interface . . . 72

7 Evaluation and Rediscovery of Attacks 75
7.1 Evaluation of Fuzzing Campaigns . . . 75
7.2 Rediscovery of Attacks . . . 78

8 Conclusions & Future Work 81
8.1 Findings . . . 81
8.2 Future Work . . . 82
8.2.1 Implementation Improvements . . . 82
8.2.2 Design Improvements . . . 83
8.2.3 New Directions . . . 84

Bibliography 85

1 Introduction

The Transport Layer Security (TLS) protocol is one of the most wildly used cryptographic protocol on the Internet. It can be used to establish a secure channel between two peers in order to exchange application data. It is used in other protocols like the HyperText Transfer Protocol Secure (HTTPS) for browsing the web. Since its first release in 1994, it has experienced an abundance of revisions, including a renaming from SSL to TLS in 1999. Features have been added, removed and changed in its over 25 years lifetime.

The specification of TLS is provided through Request for Commentss (RFCs) and started as a draft until it became an actual RFC. During this time design proposals have been discussed in the public. The drafts as well as final versions of the RFC for TLS 1.3 have been modeled formally and then verified using formal verification [10, 23]. This was the first time the protocol has been formally analyzed before deployment. The Working Group behind TLS encouraged the scientific community to contribute during the draft-phase of TLS 1.3 [23]. Even though, only the main features of TLS have been verified this represents an important step in improving the security of cryptographic protocols.

One of the challenges in designing and implementing the TLS protocol is backwards compatibility. At the same time the most secure parameters for the encrypted connection need to be negotiated. This complexity lead to various vulnerabilities in the specification as well as implementations of the protocol. If the security issue arose from the specification, then the TLS Working Group would release an erratum for the original RFC and release a protocol extension. If the issue only concerns specific implementations like OpenSSL or LibreSSL, then the maintainers of these would release patches of their source code.

Implementation Vulnerabilities Traditionally, software engineers start from a specification and refine it into a concrete implementation. This process is prone to errors. Resulting implementations are usually far from being an ideal system. And in fact the history has shown that implementations of TLS deviate from their specification like the FREAK and SKIP vulnerabilities have shown in [7]. In fact Cremers et al., raises concern that implementation mistakes are most likely the hardest class of security vulnerabilities to avoid, compared to design flaws [23]. FREAK and SKIP lead to protocol downgrades and authentication bypasses. This means compared to the well-known Heartbleed vulnerability the protocol logic which is implemented was flawed. In the case of Heartbleed faulty memory handling leads to an exposure of private data. While this is also a vulnerability of high severity, it can be classified as being a common programming mistake and not a logical implementation defect.

Fuzzing Discovering vulnerabilities in implementations like OpenSSL or LibreSSL is challenging. Fuzzing is a technique of finding vulnerabilities automatically by repeatedly executing a program under test with different test inputs. Different fuzzer designs like coverage-based grey-box fuzzing [77, 33] have been used in the past. Fuzzing stateful cryptographic protocols like TLS presents a harder challenge compared to fuzzing for example file parsers.

A TLS server expects a series of messages being sent to it. Between every message the server changes its internal state. Earlier messages, influence future messages. Furthermore, the use of cryptographic hashes, message authentication codes and encryption, challenges the fuzzer with guessing secrets. Pham, Böhm, and Roychoudhury fuzz sequences of messages, but focus on fuzzing on a bit-level. This means instead of reusing observed data gathered during the protocol execution, the fuzzer mostly changes individual bytes. The fuzzer also doesn't have the capability of decrypting or encrypting messages, which is notorious important for cryptographic protocols.

Moreover, the objective of many protocol fuzzers is to find crashes in the program under test. Many logical flaws like FREAK or SKIP do not make themselves obvious through such a crash. Instead, a more sophisticated bug oracle is required.

Symbolic-model-guided Fuzzing We employ a novel technique to test implementations of TLS automatically. We present a symbolic-model-guided fuzzer in this thesis which focuses on the mentioned challenge of finding logical flaws in TLS implementations.

Protocol verification utilizes a symbolic model and represents protocol messages as terms. These terms are built from symbolic functions and variables. In this formal model attackers are able to

- eavesdrop on messages, which means they can gain knowledge from network traffic,
- perform deductions, which means they can for example decrypt data using previously learned knowledge,
- and control the network, by being able to inject, remove or tamper with messages.

With our novel approach we lift these capabilities from a formal model to concrete implementations by designing and implementing a symbolic-model-guided fuzzer.

The fuzzer is able to mutate message terms based on coverage-based feedback. These terms are composed in a trace of messages. Interesting traces, which means traces with fit feedback, are added to a population of fit test inputs. By using this feedback loop in an evolutionary fuzzing algorithm we can explore implementations successively. This algorithm is inspired by the well-known grey-box fuzzer AFL [77]. The fuzzing procedure is supervised by a bug oracle which is able to detect security violations like authentication bypasses.

1.1 Contributions

The major contributions of this thesis are presented in the following.

Symbolic Trace Model We will introduce a symbolic trace model, which is tailored to fuzzing cryptographic protocols.

Design of a Symbolic-Mode-Guided Fuzzer Based on the concept of symbolic traces we will design a protocol fuzzer. We will also introduce domain-specific mutations which are used within the fuzzing algorithm.

Novel Bug Oracle A novel bug oracle allows us to detect security violations during the execution of TLS handshakes. Together with the oracle, we also contribute a custom instrumentation of TLS libraries to detect violations.

Reference Implementation We will contribute a reference implementation in Rust for a symbolic-model-guided fuzzer to prove its feasibility. The implementation also includes a functional TLS 1.3 client.

Evaluation Lastly, we will evaluate the protocol fuzzer by showing that we are able to rediscover known vulnerabilities in TLS. We will also present a way to visualize statistics which have been gathered while rediscovering vulnerabilities.

1.2 Outline

The findings in this thesis include that the symbolic-model-guided fuzzing approach:

- is capable of adequately modeling traces of the cryptographic TLS protocol
- including deviant attack traces,
- but requires future work in order to evaluate the framework more precisely.

The findings have been obtained by firstly covering foundations about TLS in Chapter 2 and Fuzzing in Chapter 3. After that, we introduce a formal model to represent traces in Chapter 4. We also introduce symbolic and concrete semantics which specify the design of the novel fuzzer. We discuss the introduced model further in the design phase of the fuzzer in Chapter 5. The fuzzer is called TLS Protocol Under FuzzINg (tlspuffin). Furthermore, we will highlight problems which we encountered and practical solutions to it. In Chapter 6 we present our reference implementation of tlspuffin and mention important design decisions. In Chapter 7 we present statistical data about fuzzing campaigns and discuss security vulnerabilities we were able to rediscover through fuzzing, which are usually out of scope for classical bit-level fuzzers. Finally, in Chapter 8 we draw conclusions about the results of this thesis and discuss future work.

2 Foundations of the Transport Layer Security Protocol

The TLS protocol allows clients and servers on the Internet to communicate securely by preventing eavesdropping, message tampering and forgery. It is used extensively to secure channels for various applications like the web which uses HTTPS, email, chat, or Virtual Private Networks (VPNs).

The usual attack scenario, in which TLS is trying to protect agents is in the presence of a Man-in-the-middle (MITM) attacker which is depicted in Figure 2.1. In this scenario TLS is protecting the confidentiality and integrity of the data sent between the two endpoints as well as enabling both peers to authenticate against each other. While server authentication is required, client authentication is optional in TLS 1.3. [60]

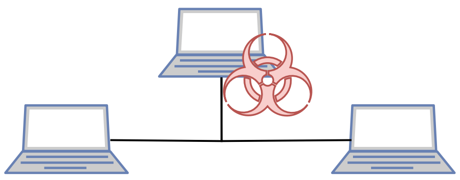


Figure 2.1: Schematic explanation of a MITM attack.

The TLS specification already went through major revisions, including a name change from SSL to TLS in 1999 [1]. This indicates that the protocol and especially its implementations are influenced by historical decisions over almost 30 years. With TLS 1.3 features were removed from the specification with the goal of simplifying it [60]. From the TLS protocol version 1.2 to 1.3 over 20 protocol extensions were removed. While there are still over 30 registered extensions left in TLS 1.3, not all are recommended and required to implement. In this thesis we limit the scope to the specification of TLS 1.2 and 1.3. In the following section we will take a detailed look at handshake of TLS 1.3. Before we can explain the full handshake of TLS 1.3, we need to define some core concepts of TLS.

Record and Handshake Protocol The TLS protocol is split into two layers. Firstly, there is the record layer which has its own protocol and the handshake layer. Because this thesis focuses on testing whether the handshake is secure, we need to take into account both layers. The purpose of the record layer is to warp TLS messages and encrypt them optionally. It can be seen as an encryption layer. The handshake protocol in the handshake layer implements the key exchange, authentication and further features.

TLS Extensions Many TLS messages include a field which specifies an array of protocol extensions. For example there are client, server, certificate, or ticket extensions. TLS uses the concept of extensions to describe different types of attachments to messages. Historically, extensions were added which overwrite other fields. For example with TLS 1.3 there is now an extension which specifies the protocol version. The concrete protocol version fields in the messages have become meaningless. Therefore, extensions present a more dynamic way to extend the protocol. At the same time extensions can become deprecated. By moving information from the message definition into extensions the protocol has gained flexibility.

Cipher Suites A cipher suite defines a set of cryptographic primitives which are used for the TLS handshake. According to the specification of TLS 1.2 a cipher suite contains the four parameters key exchange method, authentication algorithm, encryption algorithm and hash algorithm. With TLS 1.3 the specification of the cipher suite was simplified and only defines the encryption and hash algorithm. With TLS 1.3 the key exchange method is determined to be Elliptic-curve Diffie–Hellman Ephemeral (ECDHE). The authentication algorithm is defined through a TLS extension. The count of available ciphers has reduced significantly from TLS 1.2 to 1.3. An example for a TLS 1.3 cipher suite is `TLS_AES_128_GCM_SHA256` which defines that AES in GCM mode with 128-bit keys and the SHA hash algorithm with 256-bit hashes must be used.

Transcripts TLS 1.3 uses transcripts in order to provide authentication of exchanged handshake messages and the created keying material. The transcript hash is constructed by hashing the concatenation of plain text TLS handshake messages [60]. The binary representation of each message is used as input to the hash function. More formally, the transcript hash is defined as

$$\text{Transcript-Hash}(M_1, M_2, \ldots, M_n) = \text{Hash}(M_1.M_2.\ldots.M_n)\,, \tag{2.1}$$

where $M_1, M_2, \ldots, M_n$ are TLS messages encoded as bitstrings, . denotes concatenation, and Hash is a hash function which is negotiated through the cipher suite.

2.1 Handshake Overview

The goal of TLS is to establish a secure channel between two agents. This is done by several flights of messages between a client and a server. This section is structured according to phases of a TLS handshake which are shown within the dotted boxes in Figure 2.2. Each box in the figure represents a message which is being sent from the client to the server or the other way around. When we refer to TLS messages, in this thesis then we mean exactly those messages. We will describe now in detail what is happening in each phase of the protocol.

Key Exchange In the initial phase, keying material is exchanged and cryptographic parameters which will be used during and after establishing a secure channel are selected. The Client Hello message proposes a protocol version and a list of cipher suites. It also contains a set of Diffie-Hellman key shares, or a Pre-shared Key (PSK) or both as TLS extensions. Moreover, a list of additional extensions can be appended. [60]

As soon as the proposed parameters from the Client Hello are received by the server, it tries to create a more concrete proposal. For example the concrete cipher suite like `TLS_AES_128_GCM_SHA256` is selected. Amongst other parameter, the Server Hello message also sends back a key share if ECDHE is used and optionally which PSK was selected from the Client Hello. The key shares of the client and server have to be in the same elliptic curve. By using key shares instead of static keys for the key exchange, the channel demonstrates forward secrecy with respect to long-term keys. Even if long-term secrets like signature keys for certificates are compromised, the secrecy of previously negotiated session keys is not voided. The reason for this is that the key shares are ephemeral and are used solely for establishing the session keys. By purging session keys after the handshake, forward secrecy is preserved.

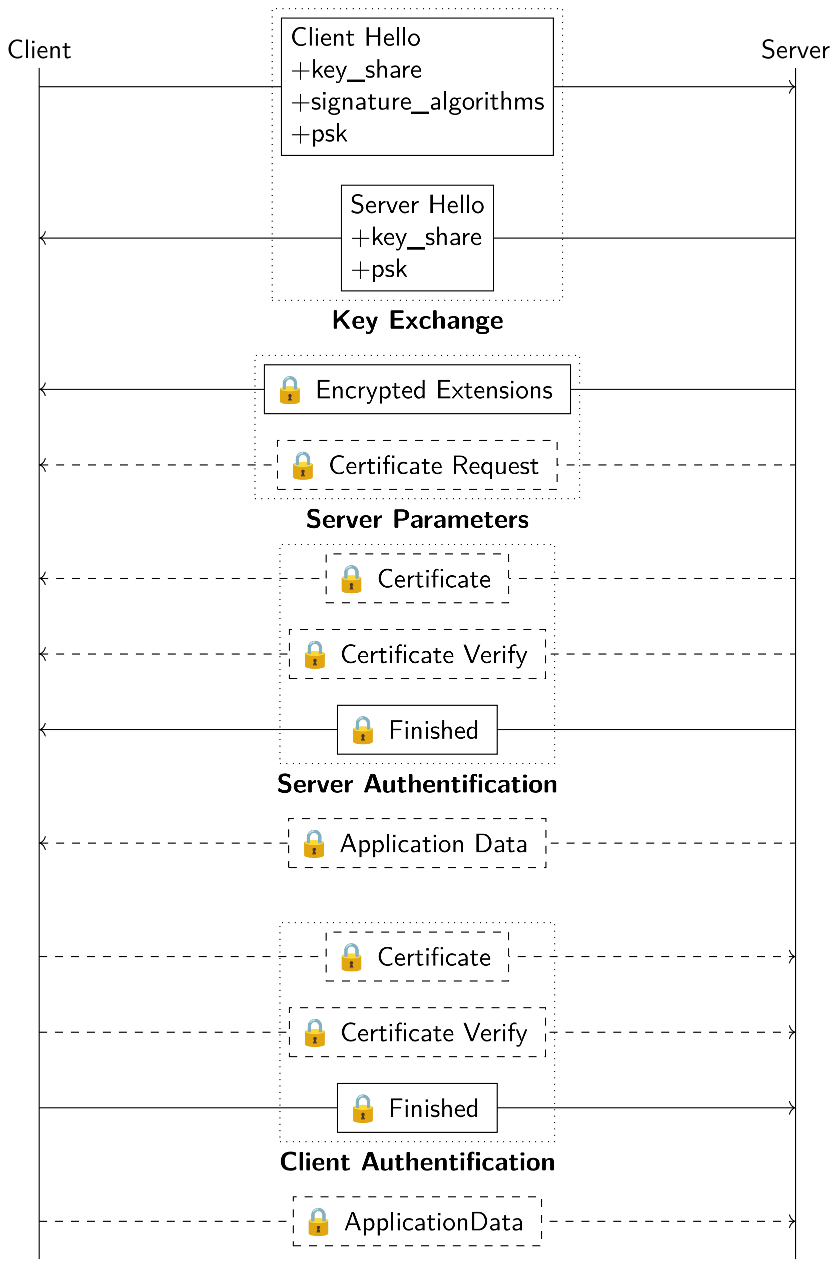


Figure 2.2: Basic TLS 1.3 protocol flow: A message sequence chart which describes the interaction of a client and server during a successful TLS 1.3 handshake which is also called the happy flow. Optional messages are denoted by a dashed arrow, whereas required ones are depicted as a straight line. Certificate and Certificate Verify messages sent by the server are required in the initial handshake but are optional in subsequent handshakes if session resumption is used (see Section 2.3). The individual phases of the protocol flow are visualized through dotted boxes around groups of messages. The pictogram 🔒 denotes that a message is encrypted.

Server Parameters The next two messages can already be transmitted encrypted by deriving the Handshake secret which is shown in Figure 2.3. The server sends the Encrypted Extensions message which can contain additional extensions which should be used. This message is available in the specification because it makes sense to send some extensions only encrypted. An example usage would be to send the name of the server via an encrypted extension. That way eavesdroppers can not determine which name the server has. The specification determines which extensions have to be sent in the Server Hello or Encrypted Extension message.

If the server wants that the client authenticates, then it has to send the Certificate Request message. By sending or omitting this message, the server can control whether the client has to send a Certificate and Certificate Verify message in the client authentication phase.

Server & Client Authentication In the TLS protocol the server is always required to authenticate itself, while client authentication is optional. This fact can also be observed in daily web browsing which uses TLS as part of the HTTPS protocol. When users observe certificate issues, then the server failed to authenticate against the client. This means that the client ruled that the certificate of the server is invalid, or that the server does not own the private signature key corresponding to the certificate.

Each of the client and server authentication phase contains the three messages Certificate, Certificate Verify and Finished. If PSKs are used, which means that the client and server already had an authenticated connection in the past, then we can skip the Certificate and Certificate Very messages, because the PSK together with the Hash-based Message Authentication Code (HMAC) in the Finished message cryptographically binds the current session to the session in which the PSK was negotiated.

Certificate The corresponding agent first sends the certificate which she wants to use. Certificates are usually sent in the X509 format which contains a public key, an issuer, a subject a validity and more fields.

Certificate Verify The Certificate Verify message contains a signature over the transcript of the whole handshake. This means that the agent proofs that she has access to the private key corresponding to the public key in the certificate and the integrity of the handshake is provided.

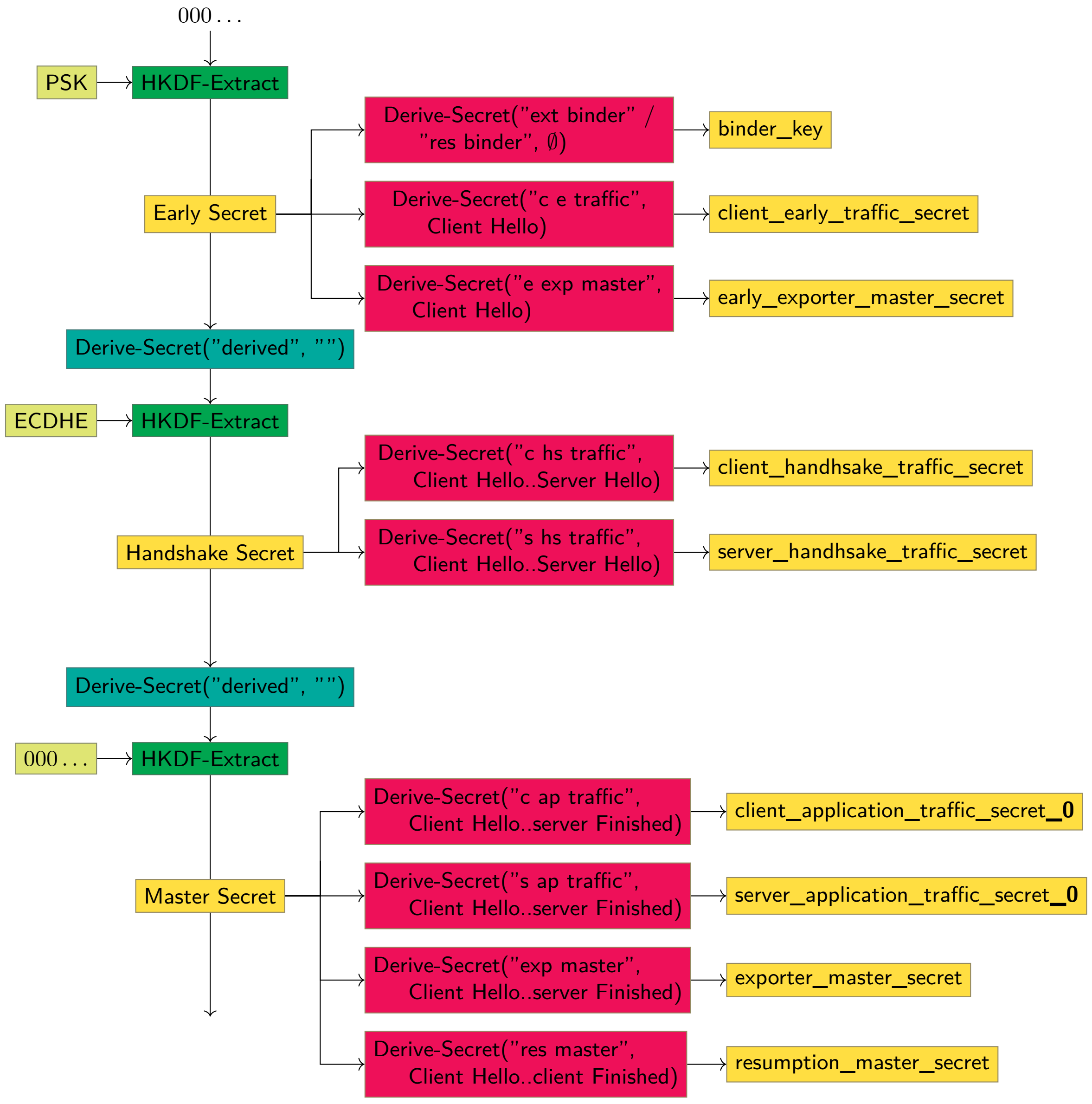
Finished The Finished message provides a HMAC over all messages sent in the current handshake. The key for the HMAC is derived from the current client or server `handshake_traffic_secret`. It also is the final message in the TLS handshake. The message guarantees the integrity of the handshake by using the client or server handshake traffic secret like they are shown in the key schedule in Figure 2.3. If PSKs are used, then this message also authenticates the handshake.

2.2 Key Schedule

We already mentioned that we derive secrets from other secrets. This section first defines which primitives are used for HMAC-based Extract-and-Expand Key Derivation Function (HKDF). For this purpose RFC 5868 defines the two functions HKDF-Expand and HKDF-Extract [43]. The HKDF-Extract(s, k) function produces a pseudorandom key from a salt s and keying material k. HKDF-Expand(k, c, L) generate a secret from a pseudorandom key k, some context c which is also known as label and the desired length of the secret L. Both functions are implemented on top of HMAC.

TLS 1.3 defines the function Derive-Secret which uses the just mentioned functions.

$$\text{Derive-Secret}(s, l, m) \coloneqq \text{HKDF-Expand}(s, \text{Context}(l, m), L)$$



000...
PSK
HKDF-Extract
Derive-Secret("ext binder" / "res binder", ∅)
binder_key
Early Secret
Derive-Secret("c e traffic", Client Hello)
client_early_traffic_secret
Derive-Secret("e exp master", Client Hello)
early_exporter_master_secret
Derive-Secret("derived", "")
ECDHE
HKDF-Extract
Derive-Secret("c hs traffic", Client Hello..Server Hello)
client_handhsake_traffic_secret
Handshake Secret
Derive-Secret("s hs traffic", Client Hello..Server Hello)
server_handhsake_traffic_secret
Derive-Secret("derived", "")
000...
HKDF-Extract
Derive-Secret("c ap traffic", Client Hello..server Finished)
client_application_traffic_secret_0
Master Secret
Derive-Secret("s ap traffic", Client Hello..server Finished)
server_application_traffic_secret_0
Derive-Secret("exp master", Client Hello..server Finished)
exporter_master_secret
Derive-Secret("res master", Client Hello..client Finished)
resumption_master_secret

Figure 2.3: The TLS 1.3 key schedule shows how secrets are derived from other secrets. There are three main secrets in TLS 1.3: Early Secret, Handshake Secret and Master Secret. Depending on the use case distinct secrets are derived from these three main secrets. The Derive-Secret and HKDF-Extract functions are defined in Section 2.2. The first argument to Derive-Secret is omitted in this figure because it is clear from the context which secret is used for the input. The input secret 000... indicates a bitstring of appropriate length consisting of only zeroes.

where s is a secret, l is a label which can be used to derive multiple distinct secrets from a single secret, m is a vector of TLS messages and L is the output length of the used hash function. The hash function is defined through the cipher suite. The function Context builds a context from the label and the vector of messages. Essentially, it concatenates the string "tls13 ", with the label l and the transcript hash of m.

Deriving Secrets In Figure 2.3 the complete key schedule is summarized. The input secrets are PSK (a Pre-shared Key) and a shared ECDHE (a Elliptic-curve Diffie–Hellman Ephemeral) key. The key schedule of TLS 1.3 requires that at least a PSK or a ECDHE key is available. If either one of those two is not available in the operating mode then a bitstring consisting of only zeroes is used. In the diagram the HKDF-Extract function takes the keying material from the left and the salt from the top. The Derive-Secret function takes keying material from the top or left and outputs a new secret to the bottom or right. It also indicates a label string like "c hs traffic" and a sequence of TLS messages like Client Hello..Server Hello, which means that all messages from Client Hello until the Server Hello message should be included in the transcript hash.

2.3 Session Resumption

Session resumption resumes already closed TLS connections. The goal of this TLS feature is to abbreviate handshakes and therefore increase performance. It avoids costly flights of communication via zero Round-Trip-Time (RTT) abbreviated handshakes. Session resumption allows for example to open multiple simultaneous sessions quickly. An abbreviated handshake works like a full handshake like shown in Figure 2.2, except that it can omit certain messages. By resuming an authenticated session, the Certificate, and Certificate Verify messages can be skipped, because the initial session was already authenticated. We introduce this feature of TLS because we will test this kind of handshake like we will describe in Section 5.3.2.

TLS 1.2 vs. 1.3 In TLS 1.2 two different ways of resuming a session are specified in the RFCs 5077 [32] and [61]. The first specifies session resumption via stateless tickets and the latter via stateful session identifiers. While the first variant does not require a state in the server, the second one requires a lookup table to be present at the server. TLS 1.3 unified this mechanism via PSKs combined with 0-RTT resumption.

PSK Establishment In TLS 1.3 a PSK is established in the initial handshake. The client then can present the PSK during the next handshake. After the initial handshake, the server can send the client a new session ticket via the New Session Ticket message [60, Section 4.6.1]. It includes a ticket lifetime, a ticket nonce, and a ticket identity which is an opaque label. It can either represent database lookup key, if the server wants stateful resumption or a stateless self-encrypted and self-authenticated value. The PSK secret is derived from the `resumption_master_secret` (see Figure 2.3) and the ticket nonce using the Derive-Secret function described inSection 2.2. This way the PSK is bound to the complete previous session because it contains its transcript and the ticket.

PSK Authentication If a client wishes to resume a previous session, it needs access to the PSK and the ticket like described in the previous paragraph. We want to create a binder which cryptographically forms a binding between the PSK and the current handshake. The binder and the ticket identity is sent within the Client Hello as an extension. The binder depends on a transcript hash of a partial Client Hello. This can be computed before creating a binder, because a partial Client Hello actually excludes the binder. This can be done by creating a Client Hello with a fake binder, and then truncating before its appearance in the bitstring of the message. Therefore, we can define the transcript like in the following equation.

$$t = \text{Transcript-Hash}(\text{Truncate}(\text{ClientHello}))$$

The client can derive from the PSK the `binder_key` like shown in the key schedule in Figure 2.3. The `binder_key` is used to compute a HMAC b like in the Finished message to authenticate the transcript hash t.

$$b = \text{MAC}(\texttt{binder\_key}, t)$$

The full Client Hello is now sent by the client and received by the server with the opaque ticket identity and binder b. If the server operates stateless, then it must persist the PSK in the ticket identity. This means when sending the New Session Ticket message in the initial handshake it encrypts the PSK with a static ticket key. In the resumption handshake the server can acquire the PSK by decrypting the identity which is sent in the Client Hello message. Because the server has access to the PSK it can verify the HMAC and authenticate the client.

For simplification reasons several checks like for the lifetime of a ticket have been excluded in this section. The interested reader can refer to RFC 8446 [60] for more details on the security of session resumption.

0-RTT As already mentioned, TLS 1.3 supports 0-RTT handshakes in combination with PSKs. This means that a client can start sending application data directly after the Client Hello. This data is encrypted using the `client_early_traffic_secret` as showing in Figure 2.3. As soon as the handshake completed, the server indicates a key change using the End Of Early Data message. It is noteworthy that 0-RTT handshakes weakens the security of TLS.

- Early data does not offer forward secrecy because it is only encrypted under keys derived from the PSK [60, Section 2.3].
- 0-RTT handshakes are not secure against replay attacks [60, Section E.5].

The first risk can be mitigated by combining PSKs with an ephemeral ECDHE key exchange. Data which is encrypted under keys which are derived from an ephemeral key enjoy forward secrecy. In this mode PSK can still be used to skip certificate-based authentication. The second risk can be mitigated by several methods like Single-Use Tickets and Client Hello Recording described in Section 8 of [60]. These tactics are external to the TLS specification and therefore not discussed here. In practice both security risks are accepted in favor of increased performance.

2.4 Security Properties and Known Attacks

Several security properties of TLS like downgrade resistance, secrecy of session keys, peer authentication or availability have been violated in the past. We will describe a few known attacks on TLS implementations in this section. The following attacks target version 1.2 of the protocol, because several logical attacks have been found in implementations which support this version. These attacks should not be possible according to the TLS specification. Mistakes in implementations lead to bugs and allowed the following vulnerabilities.

SKIP The SKIP vulnerability allows circumventing peer authentication in OpenSSL 1.0.0. If the client proposes a Diffie-Helman certificate it can skip the Certificate Verify message and therefore the key confirmation [7]. This means that an attacker can imitate the identity of the actual certificate owner, even though the adversary does not have access to the certificate's private key. The severity of the vulnerability is low because Diffie-Helman certificates are not common.

FREAK The FREAK vulnerability allows downgrading the security of TLS 1.2 in OpenSSL 1.0.0 [7]. The adversary operates as a MITM between a server and a client. The MITM replaces the secure cipher suite in the Client Hello with a more insecure `RSA_EXPORT` one. The server has to accept the weak cipher suite such that this vulnerability works. The MITM then replaces the weak cipher suite with a secure one which is supported by the client in the Server Hello message. Now, the client blindly accepts an ephemeral 512-bit RSA key exchange initiated by the server, even though a weak `RSA_EXPORT` cipher suite does not match the negotiated cipher suite from the client's perspective. Because 512-bit keys are factorizable in reasonable time, an adversary could in fact break the secrecy of the session keys. Moreover, the client and server have a different view in terms of which parameters have been negotiated. The client thinks that a secure cipher suite is used, whereas the server thinks that a `RSA_EXPORT` cipher is used.

CVE-2021-3449 This vulnerability allows to limit the availability of a TLS server running OpenSSL 1.1.1 with enabled TLS 1.2 renegotiation [25]. We did not discuss renegotiation in TLS 1.2 because it is a legacy feature of TLS. In a nutshell, renegotiation or its more secure equivalent, allows the change of connection parameters. The parameters can be changed by sending an encrypted renegotiation Client Hello after the initial handshake which includes the same HMAC from the Finished message of the initial handshake. If the initial Client Hello contains the Signature Algorithm and Certificate Signature Algorithm extensions and the renegotiation Client Hello omits the former, but includes the latter, then the server crashes because of a memory corruption.

Heartbleed The well-known Heartbleed vulnerability [31] allows the exfiltration of data from the memory of a TLS 1.2 server process executing OpenSSL 1.0.1. Because the extracted memory could potentially include various kinds of secrets, the impact of this vulnerability is not clearly categorizable. One outcome of the vulnerability is that the secrecy of session or long-term certificate signature keys can be lost. The vulnerability is triggered by sending a Heartbeat message from the client to the server. By populating a length field within the message with a large and invalid length, it is possible that the server reads memory from unintended locations and transmits it to the client. The reason for the vulnerability is a heap-buffer over-read.

3 Foundations of Fuzzing

The term “to fuzz” describes the process of testing a target program by “generating a stream of random characters to be consumed by a target program” [50]. Miller, Fredriksen, and So coined the term about 30 years ago by testing Unix utilities with random input strings [50]. A recent publication by the same author concludes that there is still need for this classical type of fuzzing today [51]. In this section we want to provide an overview of what fuzzing is and what the predominant methodology is.

3.1 Terminology

Throughout the literature various terms are used to describe concepts in fuzzing. In this section we want to define terms which are used in this thesis.

Program or Library Under Test The Library Under Test (LUT) or Program Under Test (PUT) is the piece of software which is being tested. It can either be a standalone application like a command-line program or a library which requires an initial test setup.

Testing Harness A harness handles the test setup for a given LUT or PUT. It wraps the software and initializes it such that it is ready for executing test inputs. A harness is integrating a LUT or PUT into a testing environment.

Test Input A test input is the concrete input which is given to the testing harness. Usually, it is a string or bitstring. A test input can also be encoded in a more complex format like an abstract syntax tree.

Input Space The input space corresponds to the space of all possible inputs for a software. It can be for example the space of all strings or bitstrings. It can also contain all abstract syntax trees.

Fuzzing Testing software by repeated execution with random data from an input space. Even though, the term is used mostly for security focused testing, it can also be applied when testing for other properties like safety or reliability.

Fuzzer The program which manages the fuzzing process is called a fuzzer.

Fuzzing Campaign A fuzzing campaign is an execution of the fuzzer. A fuzzing campaign starts when the fuzzer starts testing and stops when the fuzzing procedure is stopped.

Bug A bug is a state of software, in which the proper operation is compromised [40].

Fuzzing objective The fuzzing objective is the goal of the testing procedure. In most cases the objective is finding software crashes and bugs. Nonetheless, more complex objectives can be defined. For example an objective which we will discover later in this thesis is to find security vulnerabilities.

Bug Oracle The bug oracle is part of the fuzzer and decides whether a given test input leads to fulfilling the fuzzing objective.

Test Case A test case is a test input combined with metadata. Metadata is used to link additional information about executions of test inputs with the actual test inputs. For example, it is interesting to keep track about the average execution time of a test input. This can be done by keeping an up-to-date record in the metadata of the test case. Corpora contain test cases with metadata and not just test inputs.

Seed Corpus Fuzzing campaigns are usually bootstrapped using an initial set of test cases. This set is also called a seed corpus. Each test case of the corpus is called a seed. During a fuzzing campaign the seed corpus usually grows because new test cases are discovered.

Objective Corpus The objective corpus contains all test cases which fulfilled the fuzzing objective.

Coverage Each execution of a test input can potentially reach different states in the LUT or PUT. A metric to measure state exploration is called coverage. Code coverage for example measures how many lines of code have been reached when executing a test input.

Test Input Minimization The target of test input minimization is to find a test input which is smaller than the original test input but has the same coverage. What we mean with smaller is domain specific.

In the next section we want to discuss the dominant fuzzing approach, which uses a genetic algorithm to provide feedback after each test input execution.

3.2 Genetic Fuzzing Algorithm

The well-known fuzzer AFL fuzzer employs a fuzzing algorithm which is inspired by evolutionary algorithms [77, 47]. The basic idea is to maintain a population of seeds in a seed corpus. Each test case within the corpus has a certain fitness value. This can be determined by using a coverage metric to denote the fitness of a test case. An evolutionary algorithm then schedules fit test cases and applies mutations to produce offspring. Each new mutated test case is executed, and a fitness is assigned. The idea is that we only let offspring survive which is beneficial in terms of fitness. In the following we will explain this evolutionary process more closely.

Input Figure 3.1 shows pseudocode of which is run during a fuzzing campaign. Line 1 shows that we expect an initial seed corpus, an objective, and a feedback definition. We require that the initial seed corpus is not empty. The objective definition defines certain observations acquired during the execution of the software correspond to the fuzzing objective. This is usually determined by the underlying bug oracle. The feedback definition determines whether observations are interesting enough. This coincides with the previously mentioned test case fitness. Observations are interesting if they indicate a high fitness. We also start with an empty objective corpus in line 2.

Schedule In line 3 we start the fuzzing loop by scheduling one test case after the other from the seed corpus. The `schedule` function could decide to stop the fuzzing campaign eventually by returning nothing. This could happen for example, if no new test cases are added to the seed corpus.

```
 1 fn fuzz(seed_corpus, objective, feedback) {
 2   let objective_corpus = vec![];
 3   while let Some(test_case) = schedule(seed_corpus) {
 4     let offspring = mutate(test_case);
 5     let observations = execute(offspring);
 6
 7     if (feedback.is_interesting(observations)) {
 8       seed_corpus.append(offspring);
 9     }
10     if (objective.is_interesting(observations)) {
11       objective_corpus.append(offspring);
12     }
13   }
14   return objective_corpus;
15 }
```

Figure 3.1: This figure shows pseudocode of an evolutionary algorithm which can be used for fuzzing. The syntax is based on Rust, which will be used as a major programming language through this thesis.

Mutations In line 4 we mutate the test case to create offspring. Mutations can change test cases in various ways. If the test inputs are bitstrings, then one could imagine that bits are being flipped, bytes appended or the truncated. In line 5 we execute the new test case in order to generate observations. The most simple observation is the time it took to execute the mutated test case. More complex observations could include the covered lines of code within the LUT or PUT.

Population In line 8 and 11 we increase either the seed or objective corpus respectively if the observations indicate an interesting behavior of the offspring. If the offspring did not trigger a bug and therefore did not fulfill a fuzzing objective, it might still be added to the seed corpus if it represents progress. In analogy to evolutionary algorithm, this mean that we favor fit offspring. We only increase the population if the mutated test case is fit.

3.3 Dimensions of Fuzzing

In this section we want to point out how fuzzers can be categorized. For this purpose we use three dimensions along which fuzzers can be described. The next three paragraphs go over each of the dimension. In Chapter 5 we will describe where the fuzzer which is designed in this thesis fits.

Black-box, White-box and Grey-box Software testing methods can be categorized into the three major groups. The groups are called black-box, white-box and grey-box testing.

Black-box Black-box fuzzers are not able to see the internals of the LUT or PUT [47]. The software which is being tested is seen as a black-box. Only the behavior which is observable from the outside is used during a fuzzing campaign. Originally, most fuzzers are black-box fuzzers like the fist fuzzer invented by Miller, Fredriksen, and So [50].

White-box On the contrary, white-box fuzzers systematically explore the state space of the LUT or PUT. Test cases are generated by analyzing the internals of the software [47]. The term was shaped by Godefroid who used dynamic symbolic execution for test case generation [35]. The goal of symbolic execution is to reach states by solving path constraints using Satisfiability Modulo Theories (SMT) solvers. Therefore, symbolic execution introduces overhead which makes it less usable in fuzzing which depends on executing the software very often. Symbolic execution is not to be confused with the symbolic model we discuss in this thesis. They refer to distinct concepts.

Grey-box Grey-box fuzzers sit in between white and black-box fuzzers. They use internal information from the LUT or PUT, but do not rely on it. In contrast to white-box fuzzers whey also do not evaluate the complete semantics of the software under test [47].

Generation-based vs. Mutation-based Furthermore, fuzzers can be arranged on a scale which ranges from fully mutation-based to fully generation-based fuzzers. Generation-based fuzzers do not need an initial seed corpus, but instead use a description of the input space to generate inputs from scratch. Mutation-based fuzzers might not have a specification of the input space but use instead a seed corpus corresponding to valid executions. The initial seed corpus might contain test cases which reach a high coverage or use district features of the software which is being tested. Mutations are used to generate new test cases.

Note that there is no clear boundary between generation and mutation-based fuzzers. Generation-based fuzzers may also use mutations to mutate test cases and mutation-based fuzzer maybe also generate parts of the test case.

Structure Awareness Finally, the third dimension determines how aware fuzzers are of the structure of inputs. Fuzzers can either be input aware or unaware of the structure of inputs. Test inputs can be structured through a model like a grammar or a formal specification. For different use cases both approaches can make sense. Traditionally, no structure was used when fuzzing for example command-line tools like [50]. It could make sense though to help the fuzzer by providing a grammar for inputs.

Example 3.3.1. Let us finish this section by discussing the AFL fuzzer in various configurations and arranging it in the just mentioned three dimensional spectrum.

- AFL by default is a grey-box, mutation-based and not structure aware fuzzer. It uses internal code coverage data from the LUT or PUT to guide fuzzing. By default, it uses an initial seed corpus and mutates it. Furthermore, it does not require a model like a grammar.
- In [68] symbolic execution has been integrated into AFL which moves it more in the direction of white-box testing. By disabling code coverage detection we can make AFL a black-box fuzzing tool.
- In a non-default configuration AFL is able to use dictionaries from a grammar. This makes the fuzzer more structure-aware. To apply dictionaries generation-based mutation procedures can be used which generate parts of the test case.

3.4 Related Work and State-of-the-Art Protocol Fuzzing

In this section we want to give an overview of related work in the area of fuzzing cryptographic protocols. We also want to highlight state-of-the-art approaches which lead to a series of vulnerability disclosures.

Bit-level Fuzzing The predominant method for fuzzing network protocols focuses on fuzzing network packets or cryptographic primitives like the Cryptofuzz project [72]. Various implementation vulnerabilities have been discovered through bit-level fuzzing for cryptographic primitives. Even though, the mutation-based approach of Cryptofuzz is able to discover issues in cryptographic operations, it is unlikely that it will find logical issues which arise from misuse of cryptography or side-channels like described in the project's description.

The fuzzer SecFuzz goes towards the direction of the novel fuzzing approach discussed in this thesis [70]. It uses a concrete implementation of a security protocol to generate valid inputs, and mutate the inputs using a set of fuzz operators. SecFuzz introduces operators which are able to insert new messages and change textual and numerical fields within messages. It utilizes bit-level fuzzing within the mutations.

The probability that bit-level mutations lead to authentication bypasses or downgrade attacks like those covered in Section 2.4 is low. The reason for this is, that in order to trigger these vulnerabilities it is required to reach deep protocol states. We draw the conclusion that bit-level fuzzing is not adequate for fuzzing cryptographic protocols. In fact this is the major motivation for this thesis.

State Machine Fuzzing Usually protocols are implemented by following a specified state-machine. For example the specification of TLS offers a section which describes its state machine [60]. If we take a look at the OpenSSL code base as of version 1.1.1k, then we can clearly see that the implementation models the state machine from the specification.

An obvious step is to vet this state machine through fuzzing a stateful fuzzing approach [5, 58, 56]. In 2015 Beurdouche et al. discovered many flaws in the state machine of various implementations like the SKIP or FREAK vulnerability already explained in Section 2.4 [7]. Further, discrepancies between state machine implementation have been discovered which might be traced back to Postel's Robustness Principle, like reasoned by De Ruiter and Poll [26]:

> be conservative in what you do, be liberal in what you accept from others. [59]

The fuzzing methodology also has been translated to fuzzing Datagram Transport Layer Security (DTLS), which is TLS for UDP/IP instead of TCP/IP [66]. We claim that fuzzing the state machines on a TLS message level is not adequate for fuzzing cryptographic protocols like TLS. It is not possible to allow adversaries to modify the contents of messages, even though this could be required to uncover logical attacks.

TLS Testing Frameworks There are a few works which provide a framework for executing specific protocol flows in TLS. They are not fuzzers in that sense of the fuzzers introduced in the previous sections. They are more testing frameworks as they do not have the goal to execute the TLS implementation in a loop. Furthermore, the frameworks mentioned in this section are not written in a machine-oriented language like C/C++ which requires that a network stack to be available during execution. The thereby induced performance penalty indicates that the tools do not have to goal to discover new vulnerabilities, but check for known vulnerabilities to avoid regressions.

The tlsfuzzer project allows executing tests against a TLS server and check for vulnerabilities like DROWN or ROBOT [42]. The test cases defined in the framework send and expect TLS packets. The GnuTLS and NSS implementations of TLS use it to avoid regressions.

TLS-Attacker acts like a TLS client which executes traces in configurable ways [67]. It is possible to specify a trace which send and expect messages. For example, it is possible to first send a Client Hello message and then expect that the LUT sends back a Server Hello message. This allows to check for vulnerabilities like Heartbleed. TLS-Attacker, tlsfuzzer offer a solid base to create manual test cases. So far they lack a way of automatically discovering new vulnerabilities.

flexTLS is a verified TLS 1.3 implementation. It also allows describing test cases for TLS communications [8]. Like tlsfuzzer or TLS-Attacker the choice of the languages F# and F* makes it difficult to use flexTLS as part of a fuzzer. The three major application of the project are 1) implementing exploits for protocol and implementation bugs, 2) automated fuzzing of various implementations of the TLS, and 3) rapid prototyping of the new or upcoming TLS RFC drafts. The SmackTLS is based on flexTLS and allows some basic fuzzing [8]. For the already mentioned performance reasons it is unlikely that is can be used as part of a genetic fuzzing algorithm.

Towards Logical Fuzzing This thesis builds on knowledge gathered during a preceding internship project at LORIA with a similar goal [63]. The work utilized the fuzzer IJON [3] to fuzz traces. Traces consist of an abstract execution trace which creates agents, and performs a handshake. The fuzzer is able to control several fields in the in messages like `ClientHello`. The approach is lacking a method for controlling arbitrary fields.

4 Term Rewriting and Symbolic Traces

Formal methods have become an essential tool in the security analysis of cryptographic protocols. Modern tools like ProVerif [15], Tamarin [49], AVISPA [2], Scyther [24], or Maude-NPA [64] feature a fully-automated framework to model and verify security protocols. The underlying theory of these tools uses a symbolic model which originates from the work of Dolev and Yao [30]. This abstract model uses a term algebra to represent messages symbolically. In this model attackers have full control over the messages being sent within the communication network. An adversary can eavesdrop on, inject or manipulate messages. The cryptographic primitives are modeled through abstracted semantics. The reason for this is, that the focus of the symbolic model lies in finding logical protocol flaws and is not concerned with correctness of cryptographic primitives. Using this model it was already possible to find attacks in the protocol logic. The TLS specification has already undergone various analyses by these tools [11, 10, 23].

Based on the idea we are designing a fuzzer in this thesis which is guided by a symbolic formal model. In the following section we will introduce a formal model of terms and traces. This formal model will guide the design of the fuzzer in the next section.

4.1 Term Algebra

A common practice when modeling security protocols is to model messages of protocols as terms using a symbolic term algebra. Cryptographic operations are modeled by function symbols of fixed arity. Therefore, we define a signature $\mathcal{F} = \{f/n, g/m, \dots\}$, which is a finite set of function symbols together with the arity of each symbol. The arity k of a function f is defined as $\text{arity}(f) = k$. This set of functions contains both constructors $\mathcal{F}_c$ and destructors $\mathcal{F}_d$, where $\mathcal{F} = \mathcal{F}_c \uplus \mathcal{F}_d$. Examples for constructors are encryption, signatures, and hashes, which do not fail. Destructors like a decryption function on the other hand can fail. A more formal definition for failure will be given later in this chapter using the Msg predicate. Function symbols with an arity of 0 are called constants. Furthermore, we define an infinite set of variables, denoted $\mathcal{X} = \{a, b, c, \dots\}$. Variables are holes in terms which can be filled by other terms.

Example 4.1.1. This example shows a common signature for modeling standard cryptographic primitives like symmetric and asymmetric encryption. This definition will also serve as basis for Example 4.1.2 which introduces a specific protocol. It also contains a function to derive the public key from a private key and the constant 0.

$$\mathcal{F}_c = \{\text{senc}/2, \text{aenc}/2, \text{pk}/1, \text{zero}/0\}$$
$$\mathcal{F}_d = \{\text{sdec}/2, \text{adec}/2\}$$

What we are missing is atomic data to which the functions can be applied to. When modeling security protocols, we typically use the concept of names $\mathcal{N} = \{n, r, s, \dots\}$. Names can be nonces, random data, session identifiers or keys. $\mathcal{N}_{pub}$ contains names which are public and available to the attacker e.g. a session identifier or IP addresses. In order to model that an adversary has the capability of choosing random values $\mathcal{N}_{pub}$ is infinite. $\mathcal{N}_{prv}$ includes private keys of agents.

The separation between public and private names is similar like in reality. There are values in cryptographic protocols which are public and some which are private. Private names can become known to the attacker by observing it on the network.

Terms For any $F \subseteq \mathcal{F}$, $N \subseteq \mathcal{N}$ and $V \subseteq \mathcal{X}$ the set of *terms* $\mathcal{T}(F, N \cup V)$ is built from function symbols F, names N and variables V by applying function symbols. Its syntax in defined in Equation (4.1).

$$\begin{aligned} t, t_1, \ldots ::= {} & n && n \in N \\ & f(t_1, \ldots, t_k) && f \in F,\ \text{arity}(f) = k,\ t_1, \ldots, t_k \in \mathcal{T}(F, N \cup V) \end{aligned} \tag{4.1}$$

A *constructor term* is part of the set $\mathcal{T}(\mathcal{F}_c, \mathcal{N} \cup \mathcal{X})$. Moreover, we distinguish between two types of terms: protocol terms and recipes. While protocol terms which describe protocols are defined by $\mathcal{T}(\mathcal{F}, \mathcal{N} \cup \mathcal{X})$, recipes are defined by the set $\mathcal{T}(\mathcal{F}, \mathcal{N}_{pub} \cup \mathcal{X})$. Recipes are terms which allow an adversary to deduce data from available data. For example if an attacker wants to decrypt data and has access to the encryption key then he can use it to deduce the plain text. If we limit the allowed atoms and functions to $\mathcal{F}$ and $\mathcal{N}$ respectively, then we get the set of all closed terms $\mathcal{T}(\mathcal{F}, \mathcal{N})$. We also call these terms ground terms.

Example 4.1.2. A simplified Needham-Schroeder protocol will serve as an example in this section to demonstrate terms, as well as traces [54, 15]. In order to introduce the protocol we are using an informal notation which shows how messages are being sent between Alice and Bob as terms, in which $\mathcal{F}_N = \{\text{aenc}/2, \text{senc}/2, \text{pk}/1, \text{pair}/2\} \cup \{\text{sdec}/2, \text{adec}/2, \pi_1/1, \pi_2/1\}$ and $s, N_a, N_b, sk_A, sk_B \in \mathcal{N}_{prv}$. In the following we will use the notation $\langle x, y \rangle$ if we mean $\text{pair}(x, y)$. The right-hand side in the following notation represent examples for terms.

$$A \to B\colon\quad \text{aenc}(\langle N_a, \text{pk}(sk_A) \rangle, \text{pk}(sk_B)) \tag{1}$$
$$B \to A\colon\quad \text{aenc}(\langle N_a, N_b \rangle, \text{pk}(sk_A)) \tag{2}$$
$$A \to B\colon\quad \text{aenc}(N_b, \text{pk}(sk_B)) \tag{3}$$

We assume that the participants Alice and Bob already exchanged their public keys over a secure channel. The identities of both parties are modeled through their public keys. The protocol exchanges the nonces N_a and N_b between the participants respectively. These nonces can then be used as keys to send messages encrypted between the two. For example the secret s can be sent from Alice to Bob with the message term $\text{senc}(s, N_a)$. So far we have not introduced the semantics of the symbols used in these examples. This means a reader can guess what the above model means, but there is no clear formal definition given yet what a decryption or projection is. Therefore, we will introduce formal semantics in Section 4.2.

The Needham-Schroeder protocol contains an issue which allows a MITM attack such that Alice can be impersonated. This flaw has been discovered by Lowe [45]. It can be fixed by including the identity of Bob in step 2 by changing it to $\text{aenc}(\langle N_a, \langle N_b, pk(B) \rangle \rangle, \text{pk}(sk_A))$. The details of this fix will not be vetted further as the focus of this example is to model an attack.

Variable set The function $\text{vars}(t)$ describes the set of variables used in the term t. It is defined in Equation (4.2).

$$\text{vars}(t) = \begin{cases} \bigcup_{i=1}^{n} \text{vars}(t_i) & \text{for } t = f(t_1, \ldots, t_n) \\ \{t\} & \text{for } t \in \mathcal{X} \\ \emptyset & \text{otherwise} \end{cases} \tag{4.2}$$

Example 4.1.3. This example shows the result of applying the function vars on a term. The variables x and y are returned.

$$\mathrm{vars}(\mathrm{sdec}(\mathrm{senc}(h(x), y), y)) = \{y, x\}$$

Subterm set The function $\mathrm{st}(t)$ describes the set of all subterms of the term t and is defined in Equation (4.3). The size of a term t is defined as $|\mathrm{st}(t)|$.

$$\mathrm{st}(t) = \begin{cases} \{t\} \cup \bigcup_{i=1}^{n} \mathrm{st}(t_i) & \text{for } t = f(t_1, \dots, t_n) \\ \{t\} & \text{otherwise} \end{cases} \tag{4.3}$$

Example 4.1.4. This example shows the result of applying the function on a term. Every possible subterm, including individual variables and names is collected and added to the output of the function.

$$\mathrm{st}(\mathrm{sdec}(\mathrm{senc}(h(x), y), y)) = \{y, x, h(x), \mathrm{senc}(h(x), y), \mathrm{sdec}(\mathrm{senc}(h(x), y), y)\}$$

Substitution In order to manipulate terms and transform them we need to introduce the concept of substitution. A substitution σ maps variables $x \in \mathcal{X}$ to terms $\tilde{t} \in \mathcal{T}(\mathcal{F}, \mathcal{N} \cup \mathcal{X})$ by replacing the variables. This definition can be homomorphically extended to a mapping from terms to terms [57, 20]. Usually one writes postfix $t\sigma$ instead of $\sigma(t)$ to apply the substitution $\sigma = \{x \mapsto t_1, y \mapsto t_2, \dots\}$. We define the domain of the substitution as $\mathrm{dom}(\sigma) = \{x, y, z, \dots\} \subseteq \mathcal{X}$. Formally substitution is defined in Equation (4.4).

$$t\sigma = \begin{cases} f(t_1\sigma, \dots, t_n\sigma) & \text{for } t = f(t_1, \dots, t_n) \\ t\sigma & \text{for } t \in \mathrm{dom}(\sigma) \\ t & \text{otherwise} \end{cases} \tag{4.4}$$

This means, if we encounter a term which is a function, then we apply the substitution on all arguments. If the term is in $\mathrm{dom}(\sigma)$, then we can replace it with the corresponding term in the substitution. If the term is not a function and is not in the domain of σ, then we do nothing and omit σ.

Example 4.1.5. Let us provide some more intuition by supplementing the definition with an example. This example uses the substitution $\sigma = \{k \mapsto \mathrm{utf8\_decode}(a)\}$, which replaces the variable k with the binary representation of a name s.

$$\mathrm{sdec}(x, k)\sigma = \mathrm{sdec}(x\sigma, k\sigma) = \mathrm{sdec}(x, \mathrm{utf8\_decode}(s))$$

4.2 Equational Theories and Term Rewriting

Based on this definition of a term algebra we can define the semantics of terms. The semantic allows us to reason about the equality of terms. We suppose that the semantics of destructor terms are provided by a rewrite system and an equational theory defines it for constructor terms.

Equational Theories Plaisted puts it quite simple: “An equational system is a set of equations.” [57]. Formally, an equational theory E is defined by a set of pairs (t_1, t_2), where $t_1, t_2 \in \mathcal{T}(\mathcal{F}_c, \mathcal{N} \cup \mathcal{X})$. These pairs induce a relation $=_E$ on constructor terms, defined by the smallest equivalence relation $=_E$ that contains all $(t_1, t_2) \in E$, closed under substitution of variables, and under application of function symbols. This induction can be defined more precisely using following induction rules, where σ is a substitution, t, t_i and t_j are terms and f is a constructor function.

$$\frac{}{t =_E t} \qquad \frac{(t_i, t_j) \in E}{t_i =_E t_j} \qquad \frac{t_i =_E t_j}{t_i\sigma =_E t_j\sigma} \qquad \frac{\forall 1 \leq i \leq n. t_i =_E t'_i}{f(t_1, \ldots, t_n) =_E f(t'_1, \ldots, t'_n)}$$

With these rule system it is possible to prove equality of terms based on the semantics of the used function symbols, like demonstrated in the next example. By abuse of notation we write $t_1 = t_2$ in definitions of equational system instead of (t_1, t_2).

Example 4.2.1. Let us assume we want to model one of the properties of TLS 1.3 and its Diffie-Hellman key exchange which is shown in the next equation [27, 21].

$$G^{xy} \bmod p = G^{yx} \bmod p$$

Because we are not dealing with arithmetic nuances in the symbolic model, we can abstract $\bmod p$ away and focus on the commutativity of Diffie-Hellman. Even though the left and the right side of the equation are syntactically different, they both mean the very same. Therefore, we need to define its semantic using an equational theory.

$$E_{DH} = \{\exp(\exp(z, x), y) = \exp(\exp(z, y), x)\}$$

Using this equation system and by applying the above rule system we see for example that $f(\exp(\exp(z, x), y)) =_{E_{DH}} f(\exp(\exp(z, y), x))$ is true for an arbitrary function $f \in \mathcal{F}$. A proof for this can be constructed by using a proof tree.

Term Rewriting Systems A Term Rewrite Systems (TRS) R is a finite relation on terms. A rewrite rule is a pair $(\ell, r) \in R$, where $\ell \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $r \in \mathcal{T}(\mathcal{F}_c, \mathrm{vars}(\ell))$. Classically, there are a few restrictions on rewrite rules [9, 57]:

1. The left-hand side term l is not a variable and therefore contains at least one function symbol.
2. Each variable which appears on the right-hand side also appears on the left-hand side: $r \in \mathcal{T}(\mathcal{F}, \mathrm{vars}(\ell))$.
3. Function symbols on the right-hand side are constructors and not destructors, in order to avoid rules which avoid progress. By applying a rule we usually want to make the term simpler by removing destructors and not introducing them.

Usually term rewriting systems are done in a way such that the right-hand side is simpler than the left-hand side [57]. Research of TRS often discusses in which direction rules should be defined.

The semantics of terms is provided by describing how terms can be rewritten. We want to make the semantics now more precise by providing the following rule system, where σ is a substitution and $f \in \mathcal{F}$, defines how terms can be rewritten. The induced relation $\rightarrow$ is a superset of R and uses the $\rightarrow$ to notate rewrites.

$$\frac{(\ell, r) \in R}{\ell \rightarrow r} \qquad \frac{t \rightarrow t'}{t\sigma \rightarrow t'\sigma} \qquad \frac{t_i \rightarrow t'_i}{f(t_1, \ldots, t_i, \ldots, t_n) \rightarrow f(t_1, \ldots, t'_i, \ldots, t_n)}$$

By abuse of notation we are writing $\ell \rightarrow r$ when we mean that a term r can be deduced from ℓ, as well as when we mean $(\ell, r) \in R$. The above system allows the application of rewrite rules not only to whole terms, but also to subterms. Let us assume that a TRS has the rule $\mathrm{sdec}(\mathrm{senc}(x,k)) \rightarrow x$. By using the above induction rules it is possible to proof that $f(\mathrm{sdec}(\mathrm{senc}(x,s)) = x) \rightarrow (x)$.

We denote $\xrightarrow{*}$ as the reflexive transitive closure of the relation $\rightarrow$. If the right-hand side of $\ell \xrightarrow{*} r$ is no longer reducible, then we call r the normal form of ℓ. That means that there exists no tuple in $\rightarrow$ with can further rewrite r. We denote this as $\ell\downarrow$.

Confluence and Convergence A very important aspect of TRS is convergence, which means that it is guaranteed to terminate with a unique normal form [57]. Termination means that the repeated application of rewrite rules eventually terminates. We can assure convergence for a terminating TRS, by making sure that it has the property of confluence. Confluence means that distinct rewrite paths can be extended such that a single common term is reached. Therefore, whenever TRSs terminates and is confluent then it is also convergent. The rule $\exp(\exp(x,y),z) \rightarrow \exp(\exp(x,z),y)$ makes a rewrite system non-convergent. The reason for this is that this rule can be applied infinitely many times. Furthermore, convergence also requires confluence, which means that a unique normal form exists. The two rules $h(x) \rightarrow \mathrm{zero}$ and $h(x) \rightarrow \mathrm{one}$ make a TRS introduces two possibilities for $h(x)\downarrow$ [4].

As already mentioned in the introduction we want to use TRSs only for destructors. Therefore, we restrict TRS rules to destructor functions. Therefore, each rule of the TRS must have the following structure.

$$d(t_1, \ldots, t_n) \rightarrow r \qquad \text{where } d \in \mathcal{F}_d \text{ of arity } n, t_i \in \mathcal{T}(\mathcal{F}_c, \mathcal{X})$$

Equality of Arbitrary Terms We already defined equational theories only on constructors $\mathcal{F}_c$ and restricted the rules of our TRSs to include only destructors. Therefore, we use equivalence relation $=_E$ to check for equality between terms of the set $\mathcal{T}(\mathcal{F}_c, \mathcal{N} \cup \mathcal{X})$ and TRSs to compare destructor terms.

In the case of security protocol analysis, we suppose that protocols only send valid messages, which we capture using the Msg predicate [20]. We define $\mathrm{Msg}(t)$ to hold if and only if for any subterm $u \in \mathrm{st}(t)$: $u\downarrow \in \mathcal{T}(\mathcal{F}_c, \mathcal{N} \cup \mathcal{X})$. This means that after normalization, only constructor terms are left. If the normalized term still contains a destructor, then we say that the term fails. Suppose, we are left with the normalized term $\mathrm{sdec}(\mathrm{h}(x), k)$. In this case, the decryption would fail, as we modeled decrypt in such a way, that we can only decrypt cipher-text and not random data. This corresponds to authenticated encryption (AEAD). Note that it is possible to model encryption in a way that it is able to decrypt random data, which is true for block ciphers.

Finally, we can combine the $\mathrm{Msg}(t)$ predicate, equational theories and TRSs. We create a relation $\hat{=}_E$ which is true for arbitrary terms t_1 and t_2.

$$t_1 \hat{=}_E t_2 \Leftrightarrow \mathrm{Msg}(t_1) \wedge \mathrm{Msg}(t_2) \wedge t_1\downarrow =_E t_2\downarrow \tag{4.5}$$

The TRS is used to derive the normal form of the terms. Because the TRS is convergent, this is guaranteed to terminate. If the Msg predicate is true for both terms then we can be sure that the normalized forms contain only constructor terms. Therefore, we can use the induced relation $=_E$ of the equational theory E to check for equality between the normal forms.

Example 4.2.2. We continue on the simplified Needham-Schroeder example from Example 4.1.2. In deduce messages from other messages we first need to give the function symbols from $\mathcal{F}_N$ a semantic.

Firstly, Let us define a TRS R_N.

$$R_N = \left\{ \begin{array}{ll} \text{adec}(\text{aenc}(x, \text{pk}(k)), k) \to x & \pi_1(\langle x, y\rangle \to x \\ \text{sdec}(\text{senc}(x, k), k) \to x & \pi_2(\langle x, y\rangle) \to y \end{array} \right\}$$

Moreover, we define an empty equational system E_N. We can leave it empty because the Needham-Schroeder protocol does not require equivalences between terms like the Diffie-Hellman key exchange, which are not expressible through a TRS [27].

Now, we assume that an attacker owns the private key sk_E and witnessed firstly the term $\text{aenc}(\langle N_a, \text{pk}(sk_A)\rangle, \text{pk}(sk_E))$ and then $\text{senc}(s, N_a)$ during the execution of the protocol. Those terms are usually kept in a set called a frame which we will introduce later formally in the symbolic semantics in Section 4.4. We provide the attacker now with the capability of coming up with recipe terms. This means that an adversary can create terms $t \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{pub} \cup \mathcal{X})$ which allow him to gain additional knowledge.

After both observations of the above terms an adversary can check whether he can deduce information.

1. After the first observation, we can use the recipe term $t_1 = \pi_1(\text{adec}(x, sk_E))$ with the substitution $\sigma_1 = x \mapsto \text{aenc}(\langle N_a, \text{pk}(sk_A)\rangle, \text{pk}(sk_E))$ and then check whether we know N_a. In fact $t_1\sigma_1{\downarrow} =_{E_N} N_a$.
2. Following the second observation, we can use the recipe $t_2 = \text{sdec}(x, N_a)$ with the substitution $\sigma_2 = x \mapsto \text{senc}(s, N_a)$ which yields that $t_2\sigma_2{\downarrow} =_{E_N} s$ is true.

Therefore, an attacker can deduce the secret s. By applying the $\downarrow$ operator we reached the normal form which can no longer be simplified. In both cases the terms rewrite to the normal forms N_a and s Because of the reflexivity of $=_{E_N}$, both are included in the relation.

4.3 Symbolic Traces

Symbolic traces describe interactions between multiple agents in the presence of an attacker. The syntax of traces has been inspired by process calculi like the applied pi calculus [20, 13]. Firstly, we want to introduce handles which are a subset of the variables $\mathcal{H} \subseteq \mathcal{X}$. Handles are just like regular variables but are used as references to information which an adversary learned. The set of agents is denoted as an infinite set $\mathcal{A} \subseteq \mathcal{N}_{pub}$. The infinity of the set highlights that agents do not have to be created in the formal model. It is always possible to pick new agent $a \in \mathcal{A}$. The following syntax definition of symbolic traces use the handle $h \in \mathcal{H}$, $t \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{pub} \cup \mathcal{H})$ is a term and $u \in \mathcal{A}$.

$$\begin{aligned} T, R \ ::= {} & 0 && \text{(NULL)} \\ & \mid \bar{u}(h).T && \text{(OUTPUT)} \\ & \mid u(t).T && \text{(INPUT)} \end{aligned}$$

The above three rules NULL, OUTPUT, and INPUT define that there can be output and input steps in a trace, as well as a terminating step. For better readability of traces, a .0 can be omitted at the end of traces, because it is clear that every trace has to end with a NULL step. Steps can be concatenated using a . to build a trace with several steps. The length of a symbolic trace $|T|$ is defined as the amount of output and input steps. The set which contains all traces is denoted as $\mathcal{R}(\mathcal{T}, \mathcal{H}, \mathcal{A})$, where $\mathcal{T} = \mathcal{T}(\mathcal{F}, \mathcal{N}_{pub} \cup \mathcal{H})$.

Example 4.3.1. We will build on the example semantics of the Needham-Schroeder protocol of Example 4.2.2. During the execution of a protocol an attacker gains knowledge by observing messages being transmitted between agents Alice and Bob $\mathcal{A} = \{a, b\}$. The handle set is $\mathcal{H} = \{h_1, h_2, h_3, h_4\}$.

$$\begin{aligned}T = {} & a(\mathrm{pk}(sk_E)).\bar{a}(h_1)\\ & .b(\mathrm{aenc}(\mathrm{adec}(h_1, sk_E), \mathrm{pk}(sk_B))).\bar{b}(h_2)\\ & .a(h_2).\bar{a}(h_3)\\ & .b(\mathrm{aenc}(\mathrm{adec}(h_3, sk_E), \mathrm{pk}(sk_B))).\bar{b}(h_4)\\ & .a(h_4)\end{aligned}$$

The first pair of steps makes Alice receive the public key of the attacker $\mathrm{pk}(sk_E)$. The next pair of steps re-encrypts the message sent by Alice for Bob by decrypting the message corresponding to h_1 and encrypting it using Bob's public key. The following pair just forwards the message from Bob to Alice. Next we decrypt the received message from Alice and encrypt it with Bob's public key, such that we can send it to him. Finally, we just let Alice receive the last message sent by Bob.

Note that because we did not yet introduce a semantic for the traces, it is unclear which terms are outputted and correspond to the handles. The next example will provide an example which describes exactly what is happening during the execution of this trace.

4.4 Semantics of Symbolic Traces

The previous trace definition provides a precise notation of declaring traces. We will enrich this definition now with a symbolic and later with a concrete semantic. While the symbolic semantics is very close to the applied pi calculus, the concrete semantic is used in the fuzzer designed in this thesis.

We lift the definition from traces such that we can specify the state in which the symbolic process which is executing it is. Therefore, a trace is defined as $R = (T, s)$ where T is a trace define according to the previously defined syntax and $s \in \mathrm{State}$. Depending on whether we are using the symbolic or concrete semantics the set State is defined differently. Firstly, let us introduce a generic semantic with works for both models – the symbolic and the concrete.

$$(\bar{u}(h).T, s) \xrightarrow{\bar{u}(h)} (T, \tilde{s}) \qquad \text{where } \tilde{s} = \mathrm{output}(s, u, h),\ u \in \mathcal{A} \qquad \text{(OUTPUT)}$$

$$(u(t).T, s) \xrightarrow{u(t)} (T, \tilde{s}) \qquad \text{where } \tilde{s} = \mathrm{input}(s, u, t),\ u \in \mathcal{A} \qquad \text{(INPUT)}$$

The OUTPUT rule describes that given an output step $\bar{u}(h).T$ and a state s we can execute it by consuming the step and creating a new state $\tilde{s}$. The outputted knowledge is bound to the handle $h \in \mathcal{H}$. Respectively, the INPUT rule describes that given an input step $u(t).T$ and a state s we can execute it by consuming the step and creating a new state $\tilde{s}$ by using the recipe $t \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{pub} \cup \mathcal{H})$.

We use two partial functions in our semantics which have a different semantics in the symbolic and concrete semantics, namely output and input.

$$\begin{aligned}\mathrm{output}: &\quad \mathrm{State} \times \mathcal{A} \times \mathcal{H} \rightharpoonup \mathrm{State}\\ \mathrm{input}: &\quad \mathrm{State} \times \mathcal{A} \times \mathcal{T}(\mathcal{F}, \mathcal{N}_{pub} \cup \mathcal{H}) \rightharpoonup \mathrm{State}\end{aligned}$$

The semantics for each of these functions is given in the next two paragraphs, which describe the semantics in a symbolic and concrete way.

Symbolic Semantics Let us define State $= \{(\mathcal{P}, \Phi) | \mathcal{P}$ is a multiset of processes and Φ is a frame$\}$ as a set of tuples of process multisets according to the applied pi calculus and substitution frames of the form $\Phi = \{h_1 \mapsto t_1, \ldots, h_n \mapsto t_n\}$ [20]. Processes are not formally defined in this chapter. The interested reader can read in [22] about the applied pi calculus and in [20] for a complete definition including its semantics. A frame is a set of substitutions from handles to ground terms and represents a frame of attacker knowledge. We gradually extend this frame during the execution of a symbolic trace.

The previously declared input function maps from a process multiset $\mathcal{P}$, a frame Φ, an agent $u \in \mathcal{A}$ and recipe term $t \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{pub} \cup \mathcal{H})$ to a new process multiset $\mathcal{P}'$ and new frame Φ'. The output function has a similar signature like the input function. The difference is that the third argument is a handle $h \in \mathcal{H}$ instead of a recipe term. Both functions are partial and if it is not defined on some inputs we can not take the transition in the above generic semantic.

The **output** function is responsible for executing a step according to the process multiset $\mathcal{P}$ and persisting learned terms in the knowledge frame Φ. The function yields a new process multiset $\mathcal{P}'$ and a new frame Φ'. Firstly, we are selecting a process $P_u \in \mathcal{P}$ which corresponds to the agent $u \in \mathcal{A}$. If there is none, the output function is not defined for the given agent. Next we expect that the first step in the process is an output step of the form $\bar{u}(c, r).{P_u}'$, where c is a channel according to the applied pi calculus, r is a ground term and ${P_u}'$ is the rest of process P_u. If this is not the case, then the output function is not defined on the input state $(\mathcal{P}, \Phi) \in$ State. Otherwise, by executing the output function we define the frame $\Phi' = \Phi \cup \{h \mapsto r\}$, where $h \in \mathcal{H}$ and $h \notin \mathrm{dom}(\Phi)$. Next we consume the output step in the process of u by replacing $P_u \in \mathcal{P}$ with ${P_u}'$. The resulting multiset is called $\mathcal{P}'$.

The **input** function executes the process $P_u \in \mathcal{P}$ which corresponds to and is annotated by the agent $u \in \mathcal{A}$. It is responsible for using the frame in order to advance the process P_u. If there is no process P_u for the agent, then we say that this partial function is not defined for the agent u. As already declared, the input function has an additional argument $t \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{pub} \cup \mathcal{H})$ which is a recipe. We expect that the first step in the process P_u is an input step of the form $u(c, x).{P_u}'$, where c is a channel according to the applied pi calculus, x is a variable bound by the process P_u and ${P_u}'$ is the rest of process P_u. If this is not the case, then the input function is not defined on the input state $(\mathcal{P}, \Phi) \in$ State. By executing the input we consume the input step in the process P and substitute x in ${P_u}'$ with $t\Phi{\downarrow}$ if $\mathrm{Msg}(t\Phi{\downarrow})$. That means that we are replacing $P_u \in \mathcal{P}$ with ${P_u}'\{x \mapsto t\Phi{\downarrow}\}$ in $\mathcal{P}$ if $t\Phi$ is a message and does not fail. This is important because we want to avoid that terms within a process fail. This models in fact the reality in which message terms circulating in a network do not fail when evaluated. The resulting multiset is called $\mathcal{P}'$.

The symbolic semantics provide a way of expressing the meaning of traces close to the symbolic model [20]. While we will not use it in the fuzzer design directly it shows a clear link to the theoretical protocol verification using symbolic modeling. By highlighting the usage of processes in the semantics we can show a clear link to verification tools like ProVerif [14]. In the symbolic semantics, traces describe a specific flow through a protocol, guided by the processes which model the protocol. Next we will take a look at the concrete semantics which is are inspired by the symbolic ones and are actually used in the design of the fuzzer.

Example 4.4.1. This example describes an example execution of a symbolic trace using the symbolic semantics just introduced. Because the symbolic semantics are more complex than the concrete, an example showcasing a symbolic execution should also give an intuition how the concrete semantics operate.

We will build on Example 4.3.1 which declares a trace from the Needham-Schroeder protocol and on Example 4.2.2 which shows how to apply the semantics of terms. Firstly, Let us revisit the trace.

$$\begin{aligned} T = & a(\text{pk}(sk_E)).\bar{a}(h_1) \\ & .b(\text{aenc}(\text{adec}(h_1, sk_E), \text{pk}(sk_B))).\bar{b}(h_2) \\ & .a(h_2).\bar{a}(h_3) \\ & .b(\text{aenc}(\text{adec}(h_3, sk_E), \text{pk}(sk_B))).\bar{b}(h_4) \\ & .a(h_4) \end{aligned}$$

As already described in the last example, it mostly forwards messages by re-encrypting them if needed. This meant the attacker is able to eavesdrop on message without Alice or Bob noticing. During the execution of a trace the process definition $\mathcal{P}$ in the applied pi calculus is adapted in each and every step. Furthermore, the knowledge frame Φ is expanded during the execution. In each pair of the trace a new substitution is added to the frame Φ. The final state of the frame is shown below.

$$\Phi = \left\{ \begin{array}{ll} h_1 \mapsto \text{aenc}(\langle N_a, \text{pk}(sk_A)\rangle, h_1), & h_2 \mapsto \text{aenc}(\langle N_a, N_b\rangle, \text{pk}(sk_A)), \\ h_3 \mapsto \text{aenc}(N_b, \text{pk}(sk_E)), & h_4 \mapsto \text{aenc}(s, N_a) \end{array} \right\}$$

The process multiset $\mathcal{P}$ is defined through the applied pi calculus and encodes the output like described in the above frame Φ. The processes for Alice and Bob respectively describe how they behave. A process declares that an agent expects and waits for inputs or sends outputs. Both can happen conditionally depending on variables. The interested reader can read in [22] about the applied pi calculus and in [15] about the exact process definition of this example.

In conclusion, in each output step of the trace knowledge is added to the frame. In each input step, we are using the recipes defined in the trace and substitute them with the frame. Like already shown in Example 4.2.2 we can use the knowledge of Φ to deduce the secret $s \in \mathcal{N}_{prv}$ as an attacker. With this knowledge an adversary can break secrecy as well as authentication, because Bob thinks he is speaking to Alice, even though he is in fact communicating with the attacker.

Concrete Semantics In the concrete semantics we define the set State to be a set of opaque states. Each $s \in \text{State}$ represents the opaque state of all LUTs. It contains all variables of all the implementations. It also has access to all the knowledge which has been outputted by any agent. We model that the LUT is in fact a black-box by saying that the concrete semantics is provided by the implementation of the LUT.

Even though it is a black-box there is a function which exposes information from the current state. The function read retrieves the data which was outputted by the LUT corresponding to a specific agent in a specific state. Note that the function returns a bitstring as opposed to a term. This means in the concrete semantics we lose information about the structure of the output.

$$\text{read} : \text{State} \times \mathcal{A} \to \text{bitstring}$$

The **output** function is implemented in the LUT. It changes the internal state of the LUT owned by agent $u \in \mathcal{A}$. Moreover, we store the knowledge $\text{read}(\tilde{s}, u)$ which was outputted during the execution of the implementation and is available in the state $\tilde{s} \in \text{State}$ after the output function was called. We give the acquired bitstring a name via the handle $h \in \mathcal{H}$.

The **input** is also implemented in the LUT and changes internal states of an agent $u \in \mathcal{A}$. We want to use the recipe term $t \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{pub} \cup \mathcal{H})$ as an input for the LUT. Because concrete implementations do not expect a symbolic term as an input, but binary data we need a function evaluate which takes the state $s \in \text{State}$ and the recipe t. This function uses the available knowledge in the state s, substitutes the handles in the recipe and crafts a bitstring. The substitution of handles with learned knowledge is achieved through queries like described in the design chapter in Section 5.1.4. The concrete semantics of the function symbols $\mathcal{F}$ is provided by concrete implementations.

$$\text{evaluate} : \text{State} \times \mathcal{T}(\mathcal{F}, \mathcal{N}_{pub} \cup \mathcal{H}) \rightarrow \text{bitstring}$$

The bitstring can then be passed to the LUT. Both functions read and evaluate need to be available to the fuzzer as we will describe in Section 5.1.2. Both are implemented in concrete code and do not require the symbolic semantics of function symbols or the formal definitions of TRSs or substitutions.

5 Designing a Symbolic-Model-Guided Fuzzer

The fuzzer design in this chapter has the name “TLS Protocol Under FuzzINg”, or short tlspuffin. Details about its implementation can be found in the next chapter. In this chapter we want to focus on the design of tlspuffin.

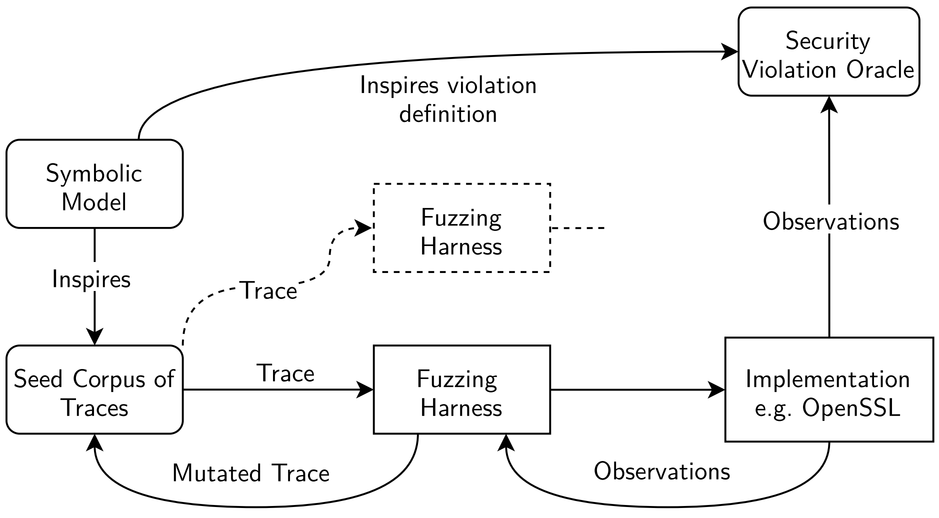


Figure 5.1: The big picture provides an informal overview of tlspuffin. Arrows are drawn in the direction of information flow. This figure should not be understood as a blueprint of the design of tlspuffin. It only captures the outline of the fuzzer’s design.

Big Picture Figure 5.1 shows the big picture of tlspuffin and provides an informal overview. As already mentioned in Chapter 4, the fuzzing approach is inspired by the symbolic model. Therefore, the reader is advised to start with Symbolic Model in the big picture. This model basically encodes the TLS specification more formally. From this model inspiration is drawn to design the initial Seed Corpus of Traces. The fuzzer multiple picks symbolic traces from the seed corpus and feeds them into the Fuzzing Harness. The harness can execute a trace within the Implementation like OpenSSL. Observations from the execution is then fed backwards in order to mutate traces and populate the seed corpus with fit offspring. The fuzzing execution is also supervised by a Security Violation Oracle, whose rule set is inspired from the symbolic model.

Fuzzer Classification In Section 3.3 we claimed that we can classify a fuzzer according to three dimensions. The fuzzing approach discussed in this chapter corresponds to grey-box fuzzing, as we have limited insights into the internals of the LUT like we will see Section 5.2. Furthermore, the approach of tlspuffin is both generation and mutation-based as we will see in Section 5.3.3. We will introduce mutations which mutate existing data and also generate new data. tlspuffin is also very structure aware. We do not employ bit-level fuzzing. Instead, the test inputs are very structured and domain-specific as we will see in Section 5.1.2.

5.1 Fuzzing Harness

A testing harness is required for most fuzzing campaigns. Depending on the program which is being fuzzed the setup of such a harness can be very different. Traditionally, especially program arguments have been fuzzed [50]. Because this task is so common the well-known fuzzers AFL and AFL++ offer a fuzzing harness which support bit-level fuzzing of command-line arguments of UNIX binaries [33]. Other programs may expect data via the standard input.

In the case of libraries it is often required to craft a harness which is specific to the library. For example image parsing require some bootstrapping and configuration before they can process data. Similarly, TLS libraries also require a setup. While it would be possible to create a harness which just passes random data to a TLS client or server, it would not be likely that the fuzzer would explore a lot of the code base. The reason for this is that cryptographic protocols operate on sequences of messages. Previous messages, influence future ones. Furthermore, in classical bit-level the fuzzer does have access to secrets which are required to craft encrypted messages. The challenge of protocol fuzzing has already been described by related work as discussed in Section 3.4.

Therefore, one of the main contributions of tlspuffin is a fuzzing harness which avoids this usual limitation of fuzzing by providing an engine for executing symbolic traces. The concepts which are essential for the design of the harness are agents, traces, and terms. From Section 4.1 we already know formally what a trace and a term is. We also know that there are names which reference protocol agents. In the following sections we will discuss the design of tlspuffin and link it to the theoretical foundations which have been established in the previous chapters.

5.1.1 Agents

In Section 4.3 we introduced symbolic traces together with the set of agents $\mathcal{A}$. Each agent corresponds to an honest protocol participant. Adversaries are not modeled through agents, instead they are modeled through the capability of intercepting, modifying or injecting messages. This capability is expressed through traces which are explained in the next section.

Each agent has an instance of the LUT and is responsible for managing it. This mean that each honest agent is running an implementation of the TLS protocol. Attackers are able to send and receive messages, but do not have access to the internal state of the TLS LUT. An agent is responsible for bootstrapping and reading down the TLS library. It also passes data to the LUT and handles errors which occur while the TLS library is processing data.

At the same time each agent also has a protocol role. In the case of TLS there are only two roles: client and server. Additionally, agents can be adjusted through a configuration which could include the maximum TLS version the library should use or the allowed ciphers.

tlspuffin supports the reuse of agents in order to support the session resumption feature of TLS 1.3 and therefore, also covering more protocol executions of the TLS protocol. The lifetime of a LUT which implements the server role needs to survive until both the initial handshake and the resumption handshake is done. The reason for this fact is that TLS server use self-authenticating stateless tickets for session resumption [60, 32]. An OpenSSL server for example generates keys which are used when creating and verifying tickets. These keys can also be rotated regularly. After restarting the OpenSSL server or rotating the keys previously issued tickets become invalid. Restarting an OpenSSL server is equivalent to recreating an agent as it reallocates all the memory used for the server. Therefore, one solution for these cases is to reuse agents without uninitializing and initializing the LUT which is described in Section 6.3. In fact, agent reuse corresponds to the usual mode in which TLS servers are operated. TLS libraries do not reinitialize their complete state for each handshake they perform, but reuse their state and context for many handshakes. Usually, servers run for a long time and only periodically regenerate ticket keys.

An alternative solution to this problem would have been to configure the LUT to use the same ticket keys every time they are initialized. The problem is that not every LUT supports this configuration and would reduce the adaptability of tlspuffin to other LUTs[1]. Moreover, avoiding time-consuming initialization improves the overall performance of the fuzzer.

5.1.2 Traces as Input Space

Every fuzzer has to define what the input for the fuzzing harness is. Usually, an array of bytes is being passed to the fuzzer. While this can provide good results when fuzzing simple command line tools, fuzzing a TLS client or server present a greater challenge because the space is very big and deep protocol states are difficult to reach [56]. Therefore, we are using traces as a high level input for the fuzzing harness. In this section we will revisit the term algebra and the notion of traces which have been introduced in Section 4.1 and Section 4.3 respectively.

```
1  let trace = define_tls_trace();
2  let context = new_context();
3
4  trace.spawn_agents(context);
5
6  for step in trace.steps {
7    step.execute(context);
8    context.execute_output_step();
9    context.check_security_violation_oracle();
10 }
```

Figure 5.2: Algorithm for executing a trace. After defining a trace definition and initializing a context in line 1 and 2, we spawn agents with their corresponding LUT in line 4. Next we loop over the input and output steps declared by the trace and execute them in line 7. After each execution we also execute an explicit output step in line 8 and check for security violations in line 9 using the oracle as described in Section 5.2.

Terms and Concrete Functions Terms are essentially trees with functions as nodes and variables and constants as leaves. When evaluating a term, there need to be concrete implementations of the used functions available. For example if an attacker wants to send encrypted text to an agent then we need to implement a function which produces cipher text based on input data and a key. The corresponding recipe term is enc_aes_128_gcm(encode_utf8(s)), where s is a string constant. The exact type depends on the programming language in which the term is represented. The names of the concrete functions is very specific. It tells us which encryption algorithm is used, as well as the text encoding schema. This is on purpose because these concrete functions are manually implemented in

[1]For example OpenSSL allows overwriting ticket keys via a callback: https://www.openssl.org/docs/man1.1.1/man3/SSL_CTX_set_tlsext_ticket_key_cb.html

a programming language. External libraries or reference implementation of the TLS protocol can be used to simplify the task of manually implementing function symbols. Concrete functions are no longer an abstract symbol like in the formal verification of cryptographic protocols [14].

Trace Execution Symbolic traces have input and output steps. Both steps drive the internal state machine of an agent forward like previously described by the input and output functions in the semantics of traces in Section 4.4. This means the LUT is being instructed to process a byte array resulting from evaluating a recipe and produce a new output and move an initial opaque state s to the state $\tilde{s}$. The states $s \in \text{State}$ and $\tilde{s} \in \text{State}$ have an opaque structure and depend on the LUT. They exist in order to describe the LUT as a black-box based on which the output of the LUT is influenced. We introduce now an algorithm in Figure 5.2 which is used to implement the concrete semantics given in the trace semantics in Section 4.4.

Firstly, we construct a trace and initialize a context which will collect all the data outputted by the LUTs in lines 1 and 2. Next, we spawn the agents required by the trace in line 4. The references to the concrete LUTs are stored in the context. Then, we execute each step which is defined in the trace one after the other in line 7. This can either be an input or output step. Finally, we also execute an additional output step in line 8 and check for security violations in the next line. The reason for automatically outputting data is described further below. The security violation oracle is introduced in Section 5.2.

The next two paragraphs revisit the concrete semantics introduced in Section 4.4 and link it to the execution of steps as it can be seen in the above pseudocode in line 7. A `step` in the statement `step.execute(context);` in line 7 corresponds either to an input step $u(t)$ or output step $\bar{u}(h)$, where $u \in \mathcal{A}$, $t \in \mathcal{T}(\mathcal{F}, N_{pub} \cup \mathcal{H})$ and $h \in \mathcal{H}$. We expect that the declaration of the steps within a trace specifies which agent u should execute the step and also either the recipe term t or the handle h.

Input Steps If `step` is an input step of the form $u(t)$, then we evaluate the recipe term t by executing all concrete functions of the term. We start with the most inner subterm of the recipe and recursively execute all functions until we get the result of the whole term. Then we need to make sure that the final result is serialized as bitstring. The resulting bitstring is used to drive the state machine of the LUT owned by the agent u forward.

Output Steps If `step` is an output step of the form $\bar{u}(h)$, then we record the outputted knowledge of the LUT and persisting it for reuse in input steps. In the symbolic semantics of traces in Section 4.4 this is done by adding a substitution $h \mapsto t$ to the frame Φ when the output step $\bar{u}(h)$ is executed. Unfortunately, a LUT is not outputting terms, but an opaque bitstring as already defined in the concrete semantics in Section 4.4. Therefore, the opaque output of the LUT is not revealing which secrets or functions were used to create the output. In the symbolic semantics we are able to reason about output terms whereas in concrete semantics we have to parse bitstring and deconstruct TLS messages.

Moreover, when defining input steps of traces which use outputted data we need a robust way of referencing it like we will mention in Section 5.4.1. The naïve way of assigning each piece of data an auto-incrementing index is not very robust because small modifications like removing or adding the optional Change Cipher Spec message in a TLS 1.3 trace could change all successive indices. Therefore, in Section 5.1.4 queries are used to reference outputted data and substitute the handles $h \in \mathcal{H}$.

By design tlspuffin automatically executes an output step after each input step. Usually after sending data to a LUT, it also returns data. Because this rule is true for the TLS protocol we can embed this rule directly in the execution of a trace. The only exception in which it makes sense to add an output step manually is at the very start of a trace. At this position there is no preceding input step, and therefore no automatically added output step. By adding an output step at the start of the trace we can let a client initiate a TLS handshake by asking it to output the first message.

Therefore, there are two major challenges which arise from the fact that we want to process the output of the LUT.

- We need to make the knowledge which is represented in frames available such that they can be used in terms.
- We need a way to reference and query previously learned knowledge.

The next chapters describe how we can extract knowledge from the output of LUTs and later on query it.

5.1.3 Auto Extraction of Knowledge

A protocol defines clear rules of communication in order to exchange information. Especially in cryptographic protocols basing responses on previously send and received messages is very important. For example every key exchange protocol like the infamous Needham-Schroeder protocol requires that public keys are exchanged before subsequent protocol messages are exchanged [54]. Therefore, traces need a way of referencing previously received data. Previously sent, data can be reproduced by executing the same trace step as previously. Therefore, on the one hand tlspuffin gathers information during the execution of a trace and on the other hand it uses gathered knowledge to supply data to LUTs. Whereas in the symbolic model this is described through a frame Φ in the concrete model we store the bitstring knowledge in a global state, available to all agents and the attacker. This global state is also called a context.

The problem with bitstrings emitted by LUTs is that they do not have an inherent structure like message terms in the symbolic model of Section 4.4 have. Each exchange message can be rewritten using a TRS. In the concrete semantics we have to work with bitstrings instead of terms.

Notable, there are two ways of providing structure to bitstrings. We can either:

- introduce destructor symbols which deconstruct a bitstring, or
- add a feature which automatically extracts information from bitstrings up to a certain granularity.

The first option requires implementing destructors on bitstrings. In order to cover many deconstruction cases a lot of manual implementations are necessary. The second option can be applied to bitstrings in a more generic way. Because we know that each bitstring can be parsed into a TLS message, we can in fact automatically extract a deep structure. Each TLS record received from a LUT is parsed into structured data. This structured data is then disassembled into pieces of knowledge which are added to a list which is kept alive until the execution of the trace finished. For example a record which includes a Client Hello message is split into its fields like the session ID, protocol version, random data, cipher suites and extensions. It is good if the structuring is as deep as possible as we want to allow the fuzzer to access any data an attacker could access. This requires a good parser for TLS records which is able to disassemble bitstrings into structured TLS messages.

Granularity A meaningful balance must be found between fineness and coarseness for auto extraction. The granularity of the auto extract has to match the symbolic capabilities of the attacker. If the auto extraction is too fine or too coarse then the term algebra which operates on the extracted knowledge is no longer applicable. The next two paragraphs discuss the just mentioned trade-offs between fineness and coarseness.

If the disassembly is too fine, then knowledge becomes difficult to refer to. An example is to add each byte of the session ID individually to the knowledge. It is very unlikely that a byte is meaningful on its own. Usually a session ID is only interesting as a whole. In the case of the supported cipher suites it could make sense to add each cipher individually, as it is clear that all cipher suites which correspond to a TLS client are offered cipher suites. A cipher suite which has been gathered from a TLS server is most likely the proposed cipher suite.

If data is disassembled too coarse then we might be unable to refer to knowledge we are interested in. While it is possible to refer to a structured TLS message and then do the extraction of knowledge in the concrete implementations of the function symbols, this has a major downside: The function would require a very specific function argument which corresponds to a structured TLS message instead of the knowledge it actually operates on. Therefore, the function becomes less reusable. For example, let us assume that there is a concrete function which sets the session ID in the Client Hello message to zero. One way to implement the function would be to expect a Client Hello as a function argument. The other possibility is to expect the session ID itself. While the first one requires a very coarse extraction of knowledge the second one requires a more fine extraction as the session ID must be available in the gathered knowledge. The second option has the advantage that zeroing a session ID can be reused in the context of a Server Hello message which also has a session ID field.

In a nutshell, the auto extraction feature of tlspuffin implements destructors by using a single extraction function which is applied to all incoming data. This is beneficial because we can exclude many destructors in the term definitions as well as avoiding implementing them. Nonetheless, the design of tlspuffin does allow modeling destructors as functions. In fact, tlspuffin includes destructor functions to decrypt data or to extract an extension by name from the list of extensions. Note that these functions can fail as defined in Section 4.1.

5.1.4 Knowledge Queries

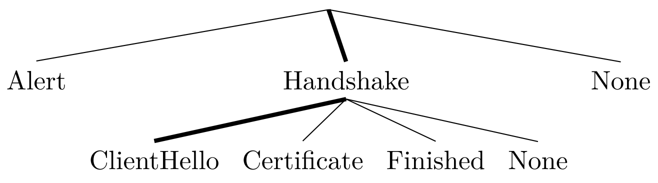
We discussed in the last section how knowledge is gathered and that we can refer to it. This reference to knowledge is created using queries. Instead of using handles like in the initial definition of traces in Section 4.3 we want to use queries to reference knowledge more robustly. We denote queries with the letter q instead of the usual letters $h_1, \dots, h_n \in \mathcal{H}$ and provide the definition of the query in natural language. In the implementation chapter in Figure 6.2 we will provide a grammar for queries, which is not yet introduced as it is implementation dependent. Queries feature four parameters, namely the **Name of Agent**, **TLS Message Type**, **Programming Language Type** and **Index**.

Name of Agent The name of the agent defines from which agent the knowledge originates. Usually there is a client and a server agent. Including which agent produced knowledge is important as there could be an ambiguity between the session ID of the Client Hello and Server Hello.

Example 5.1.1. Let us suppose an exemplary symbolic trace $T = \bar{c}(h_1).s(\text{ClientHello}(q))$, where $T \in \mathcal{R}(\mathcal{T}(\{\text{ClientHello}\}, \{h_1\}), \{h_1\}, \{c\})$ and q is defined as a query which matches knowledge by agent c. The function symbol ClientHello constructs a Client Hello message. If we assume that the agent c implements the TLS specification, then it most likely outputs a full Client Hello message.

TLS Message Type The TLS message type specifies the type of the message from which it was extracted and is derived from the TLS specification for TLS 1.3 [60] and 1.2 [61]. As we also want to allow queries to match any message type, this parameter is actually optional. On the first layer of the nested type it is possible to match either a Change Cipher Spec, Alert, Heartbeat, Application Data or Handshake message types. The former message types do not allow a nested subtype. The last message type allows to define a subtype which denotes the kind of handshake. The most common kinds are Client Hello, Server Hello, Encrypted Extensions, Certificate, Certificate Verify and Finished like shown in Figure 2.2. Like in the first layer the handshake type is also optional. In this case any Handshake kind is matched. For example the message type Handshake with the kind Client Hello matches the query with a message type Handshake without a kind. The same message type does not match the query of with message type Application Data.

Example 5.1.2. The tree below represents a TLS message type as a path through a tree. For simplicity the tree is not complete. The finite set of all paths through this tree represent all possibilities for message types. The bold connections show the path to a type which matches a Client Hello message. Let us revisit the exemplary trace defined in Example 5.1.1. A query which uses this type will only match messages of type Client Hello and therefore will only match knowledge which originates from an output step of which outputs a Client Hello. Most likely this is the case for the first step $\bar{c}(h_1)$. Note that the definition of a trace does not describe which message should be emitted. The LUT can decide on its own what should be emitted during runtime. Because of this fact, queries are an important method to reference knowledge robustly.



Programming Language Type The programming language type can directly be derived from the knowledge data via reflection of the implementations programming language. As we will see in Section 6.1 we are using Rust for our implementation. Therefore, one requirement for an implementation language is that it is typed and supports some kind of runtime type refection. During compilation the compiler of Rust assigns every type an identifier which allows equality comparisons. This identifier can be used to compare types for equality during runtime.

Example 5.1.3. Let us assume that we want to query knowledge with the programming type of a session ID. In the exemplary trace defined in Example 5.1.1 the initial output step most likely emits a session ID because it should be a Client Hello message. With the auto extraction feature mentioned in Section 5.1.3 the initial step $\bar{c}(h_1)$ should make it possible to retrieve the session ID of the client by using a query q which matches only knowledge with the programming language type of session IDs.

Index If multiple pieces of knowledge match the above three criteria then the index is used to select at most one. This is required for example if we try to query the 2nd encrypted message sent by the server agent. Encrypted messages from a single agents are expected to have the same agent name, message type and programming language type. The reason for this requirement is that almost all messages in the 1.3 protocol version of TLS have the type Application Data and are therefore encrypted. Only the Client Hello and Server Hello represent an exception.

Example 5.1.4. The exemplary trace defined in Example 5.1.1 most likely emits a Client Hello the initial output step. A Client Hello contains multiple cipher suites. In order to query multiple cipher suites it is possible to use an index of 1, 2, or 3 to match the second, third, or fourth available cipher suite.

Specificity With the above definition of queries it is possible to match multiple pieces of knowledge and then select a single one by the required parameter *index* which is often set to 0 to choose the first one. This requires a partial order on the list of result. The order is defined through the specificity of message types. A TLS message type of Application has a lower specificity than a Client Hello message. Essentially, each instance of a message type is defined by a path through a tree which represents all possible types. An example for such a tree is shown in Example 5.1.2. At every level of the tree there is a None option which matches any type. The specificity is defined by the length of the path.

The list of results is ordered in ascending order such that the knowledge with the lowest specificity has index 0. The sorting algorithm has to be stable such that knowledge with equal specificity is not reordered. The initial order of the results corresponds to the order of appearance. Moreover, knowledge is extracted in the order in which the structured data is parsed.

Knowledge with higher specificity can be queried by increasing the specificity of the query. This above described algorithm was developed while declaring traces for TLS 1.2 and 1.3 and essentially matches the pattern of the query against available knowledge. It meets the need of TLS, but it is unclear to which extent this pattern matching suffices other protocol specifications. Future work could improve on this situation by testing the current algorithm on other protocols and by generalizing the pattern matching.

Finally, it is possible that queries do not succeed. For example if we try a query for Application Data in the initial step of a trace will definitely fail because we need to send plain text data before sending encrypted data. This is also true for session resumption.

5.2 Security Violation Oracle

Traditionally, fuzzing has been used to discover memory related bugs. This type of bug is very easy to detect because operating systems clearly signal users that memory access failed by stopping the process with the exit code corresponding to a segmentation fault. This does not require instrumentation of the program. A bug oracle which observes such crashes, can detect for example if the LUT dereferences a code pointer with an invalid value [47]. Not all memory related issued can be detected reliably by observing signals of processes. It is often required to instrument a program to find the origin of memory corruptions. A well-known tool which can detect more memory related issues like use-after-free bugs is called AddressSanitizer (ASAN) (see Section 6.7) [65]. Inspired, by these classical bug oracles we designed an oracle which detects security violations. The reason is that we want to go beyond detecting simple memory issues with tlspuffin. We want to be able to detect logical flaws in the implementation of protocols like FREAK[7]. Firstly, we need to define which traces are security violations. Based on this definition we can instrument the LUT if necessary and let the oracle rule whether an execution is a violation.

Availability The already mentioned crashes related to memory issues correspond to denial of service attacks if the operating system can not continue with the execution of the LUT. A denial of service attack is defined as the violation of the goal that a system should be available to legitimate users [18]. By executing a malicious trace it is possible to crash the process which executes the LUT and therefore make it unavailable to legitimate users. Usually, applications which run a server and use a

TLS library should be aware of this possibility and should automatically restart the server in case it crashed. Nonetheless, this crash would reset the state of the server and also lead to downtime. Even though the TLS 1.3 specification does not mention the just introduced security property of availability, it introduces several other properties for the handshake layer. There are also properties for the record layer which are important for the security of the protocol, but are out of scope for the oracle discussed in this section.

Security Violation Oracle In order to introduce a bug oracle which is able to check for more advanced security violations we introduce the notation of claims. Similarly, to the AddressSanitizer [65] code sanitation tool we introduce an instrumentation together with a security violation oracle to decide whether an execution of a trace represents a security violation during runtime. A security violation oracle is defined as being a bug oracle which is able to check for security violations, as opposed to the usual oracles in fuzzing which mostly check for memory related issues.

This can be done by instrumenting the LUT and emitting claims while a handshake is being performed. The instrumentation of the LUT is done on the source code or binary level, even though it is simpler to instrument a program if the source code is available [29]. Therefore, we will also discuss the advantages of using an open source LUT in Section 6.2. Instrumentation can even happen during the runtime of a program which is common in languages like Java or JavaScript which both usually run in a runtime environment.

The instrumentation must be able to expose private data from the LUT like session keys, key exchange secrets, transcript hashes or session parameters including the used cipher suites. tlspuffin collects this data as claims. Each claim has a type and corresponding information attached. By default, tlspuffin expects that a claim is emitted by each agent after receiving or sending a TLS message.

Example 5.2.1. Firstly, let us formalize a list of claims l_a by agent $a \in \mathcal{A}$ as a finite sequence of tuples $(c_{\text{id}}, t, k, s_{\text{used}}, s_{\text{best}}) \in C$, where C is a relation over all claims, $c_{\text{id}} \in \mathcal{N}_{pub}$ is a unique connection identifier, $t \in \mathcal{N}_{pub}$ is the type of claim, $k \in \mathcal{N}_{prv}$ is the handshake master secret and $s_{\text{used}}, s_{\text{best}} \in \mathcal{N}_{pub}$ are cipher suites.

s_{used} is the currently used cipher suite, which can be undefined, if it is too early in the handshake. This is denoted by a dash. s_{best} is the best cipher suite which is available which the LUT supports. An example instantiation of a claim sequence emitted by a LUT is shown below. c_{id} can be derived from the log of TLS transcripts or a PSK and is usually public [10]. The c_{id} is only available after both a client and a server agent sent a Finished message [10]. Therefore, we also denote a missing c_{id} through a dash.

$$l_a = \begin{pmatrix} (-, \text{CLIENT\_HELLO}, k, -, \texttt{TLS\_AES\_256\_GCM\_SHA384}), \\ (c_{\text{id}}, \text{FINISHED}, k, \texttt{TLS\_AES\_256\_GCM\_SHA384}, \texttt{TLS\_AES\_256\_GCM\_SHA384}) \end{pmatrix}$$

In Section 2.4 we already mentioned several security properties of the TLS protocol, out of which tlspuffin is able to validate several using the just mentioned concept of claims. The next two paragraphs will go over them.

Unique Channel Identifier The security violation oracle offers the possibility to check if a client and server which established a secure channel use the same parameters. This property is called “Unique Channel Identifier” [10]. A slightly less strong property named “Same session keys” is defined in the specification of TLS 1.3 [60]. The difference is that instead of only comparing the session keys, we also compare other connection parameters.

We firstly require a list of claims of each LUT like introduced previously in Example 5.2.1. We also need an agent with a client and one with a server role. Based on sequences of claims l_a and l_b by both agents respectively, a security violation oracle can decide whether there is a security violation or not. The next example shows how the security oracle is able to check for the usage of the same parameters.

Example 5.2.2. Let us assume we have two sequences of claims l_a and l_b by two agents a and b. We can check whether all claims of l_a use the same parameters as in l_b by using the predicate $\text{sameParameters} \in C \times C$. In this predicate we can for example compare the protocol version, key exchange protocol, encryption algorithm, hash algorithms, elliptic curves, public key shares, session keys or PSKs [10].

The predicate $\text{sameType} \in C \times C$ is true for two claims which share the same type according to the tuples defined in Example 5.2.1. For example the tuples of type CLIENT_HELLO and FINISHED do not have the same type, whereas two CLIENT_HELLO claims have the same type.

We also introduce the predicate sameConnectionID which is true for two lists of claims like l_a and l_b, if and only if all claims of l_a and l_b have the same c_{id}. Claims of either sequences which do not yet have a c_{id} are ignored in the predicate. This corresponds to the *cid* from [10], which is computed from keying material like session keys. Usually, this ID is publicly available and created from session data provided from both a client and a server.

During a handshake each agent emits multiple claims, which contain keying material. We expect a client and server to use the same parameters throughout a handshake if they actually are in the same session according to the sameConnectionID predicate.

$$\text{sameConnectionID}(l_a, l_b) \to \forall c_a \in l_a, c_b \in l_b. \neg\text{sameType}(c_a, c_b) \vee \text{sameParameters}(c_a, c_b)$$

Downgrade protection for Cipher Suites Furthermore, it is possible to detect downgrades to a certain extent. Via the already mentioned sequences of claims we can also detect whether two LUTs selected the best available cipher suite. Compared to the previous property of Unique Channel Identifiers, we want to extend it by not only checking whether the *same* parameters are used, but also whether the *best* parameters are used.

According to the TLS specification a handshake does not have to use the most secure cipher suite. Client and server only have to agree on a specific one. In fact the specification does not even include an order on cipher suite.

In practice TLS implementations specify an order on cipher suites and try to choose the best one available, which usually is the most secure one. Let us define the best cipher suite as the first element of the cut of the list of available cipher suites by the server and client, which are ordered in descending security.

By exposing the best cipher suite available and the actually used cipher suite through claims of the client and server we can check whether the handshake used the best one. If the handshake succeeded with a different cipher suite than the best cipher suite then we witnessed a downgrade attack. An example for a downgrade attack is FREAK [7]. This attack allows an adversary to downgrade the key exchange, such that 512-bit RSA keys are used which are factorizable in a timely manner. Therefore, the consequence of the downgrade is that the confidentiality of the handshake is broken.

Example 5.2.3. Let us assume again we have two sequences of claims l_a and l_b by two agents a and b like in Example 5.2.1. We also reuse already defined predicates from Example 5.2.2.

We also check whether all claims of l_a and l_b use the best available cipher suite. The best available cipher suite can be determined by the function $\text{bestCipherSuite}(l_a, l_b) \in \mathcal{N}_{pub}$. Because claims contain the best cipher suite which is available, we can extract them from sequences of claims. Server and client agents create a claim about which cipher suite which is the best one available from their perspective. Even though, this data is not included in any TLS messages we can get this information from agents through their claims.

We also introduce another predicate $\text{usesCipherSuite} \in C \times \mathcal{N}_{pub}$ which returns true if a claim uses the provided cipher suite. If a claim does not yet contain a used cipher suite because none was negotiated so far, then we this predicate is also true.

$$\begin{aligned}\text{sameConnectionID}(l_a, l_b) \rightarrow & \forall c_a \in l_a.\, \text{usesCipherSuite}(c_a, \text{bestCipherSuite}(l_a, l_b)) \\ & \wedge \forall c_b \in l_a.\, \text{usesCipherSuite}(c_b, \text{bestCipherSuite}(l_a, l_b))\end{aligned}$$

In theory this downgrade check could be extended to more negotiated parameters than just cipher suites. For example, we could verify that the best elliptic curve or signature algorithm is used. This is out of scope for this thesis due to time constraints.

The next two security properties are out of scope of this thesis in terms of a precise design and implementation. Nonetheless, we provide a theoretical blueprint for designing the violation checks.

Peer Authentication Theoretically, it is also possible to check for authentication bypasses, by providing a client or server with a certificate without the corresponding private key. If a client succeeds to authenticate against a server or a server succeeds to authenticate against the client without possessing the required private key then there is a security violation.

An example vulnerability is CVE-2015-0205 which is also called SKIP [7]. By skipping the Certificate Verify message in the TLS protocol, it is possible to circumvent the certificate verification implemented in the server. The client sends its certificate in the previous Certificate message, but does not prove the ownership of the corresponding private key by sending the Certificate Verify message. The server is fine with skipping the Certificate Verify message and determines the client to be authenticated. The security violation check in the oracle is quite simple: If an attacker in server or client role authenticates against its corresponding peer without having the corresponding private key to the submitted certificate, then there is a security violation. Successful authentication means that the handshake was executed without errors even though the peer requests and requires authentication.

Secrecy of the Session Keys Secrecy is more difficult to prove in the case of protocol fuzzing. It is required to verify that data which is private to the LUTs is not leaked. Symbolic protocol verifiers like ProVerif which use the applied pi calculus can prove secrecy automatically for protocols for an unbounded space of messages and amount of sessions [13].

Unfortunately, we can not directly apply the technique to fuzzing. The reason for this is that LUTs do not output terms, but opaque data like described in the concrete semantics of Section 4.4. Even though we can parse this data to structure the output, we can not vet the term structure and deduct knowledge from a set of messages. The following example gives more insight on the difficulty of the task to detect secrecy violations.

Example 5.2.4. Let us suppose for example that a LUT includes the session's master key in a plain text message. In a symbolic model it is possible to notice that a leak has occurred because it can deduce the master key from the term which represents the plain text message. The challenge for the fuzzer is to understand the structure of the plain text. The task is even more difficult if the bits of the exposed master secrets are flipped using a function flip/1 $\in \mathcal{F}$. A symbolic semantic with the term rewrite rule $\text{flip}(\text{flip}(k)) \rightarrow k$ knows that by applying flip again it can deduce the secret $k \in \mathcal{N}_{prv}$. The tlspuffin fuzzer is not able to know this because it has no way of knowing that the plain text message contains $\text{flip}(x)$, where $x \in \mathcal{X}$. For the fuzzer the plain text message may only contain a bit string. A potential solution to the problem would be to statically or dynamically instrument the LUT in order to retrieve a term structure of the output. This presents a major challenge as TLS implementation like OpenSSL or LibreSSL contain legacy code and are not structured uniformly.

The other security properties mentioned in the specification of TLS 1.3 like the uniqueness of session keys or forward secrecy are not investigated by this thesis.

5.3 Fuzzing Algorithm

There are many techniques for fuzzing targets. Each approach has different strengths and weaknesses. There is also a lot of research in the field of fuzzing and the interest is growing rapidly [47]. A lot of novel ideas for scheduling test cases, innovating mutators or improving feedback in genetic-based fuzzers have shown promising results in the recent years. Yet a unifying framework which allows evaluation of fuzzing performance with different parameters was missing.

The AFL++ project noticed this and tried to bundle implementation of multiple research papers [33]. The authors even went one step further and generalized their implementation by designing a library which is called LibAFL and allows users to assemble fuzzers from generic building blocks [46]. In this section we want to explain the available building blocks of LibAFL, how they interact with each other, and which implementations we chose for designing the fuzzing algorithm for tlspuffin.

The structure of the fuzzing algorithm is showcased in Figure 5.3. It consists of three parts, namely a **Fuzzer**, an **Executor** and a **State**. Firstly, we want to prove a quick overview over these components.

- The Fuzzer selects test cases and sends them for execution to the executor.
- The Executor is observing information from an instrumented LUT during execution of the fuzzing harness. The origin of the information is either from side-channels like the execution duration, or through direct results of the execution.
- The State component is collecting test cases in the seed corpus if they are interesting according to the observations and the feedback definition. If a test case with its observations is interesting according to the objective definition, then we add it to the objective corpus.

The rest of this section goes over all buildings blocks and discusses details about the specific implementation chosen for tlspuffin. Firstly, let us define what a test case is. Similarly, to what we already covered in Section 3.1 a test case is the input which can be executed in order to achieve a specific behavior of the LUT. In this context a test case consists of a trace like defined in Section 5.1.2 which can be executed and metadata. For each trace metadata can be created and attached. An example for metadata could be the average execution time of the trace. The metadata can then be used at various steps in the fuzzing framework like for example the scheduler who is responsible to select the next test case to be executed. A scheduler can then for example prefer traces which took short time to execute.

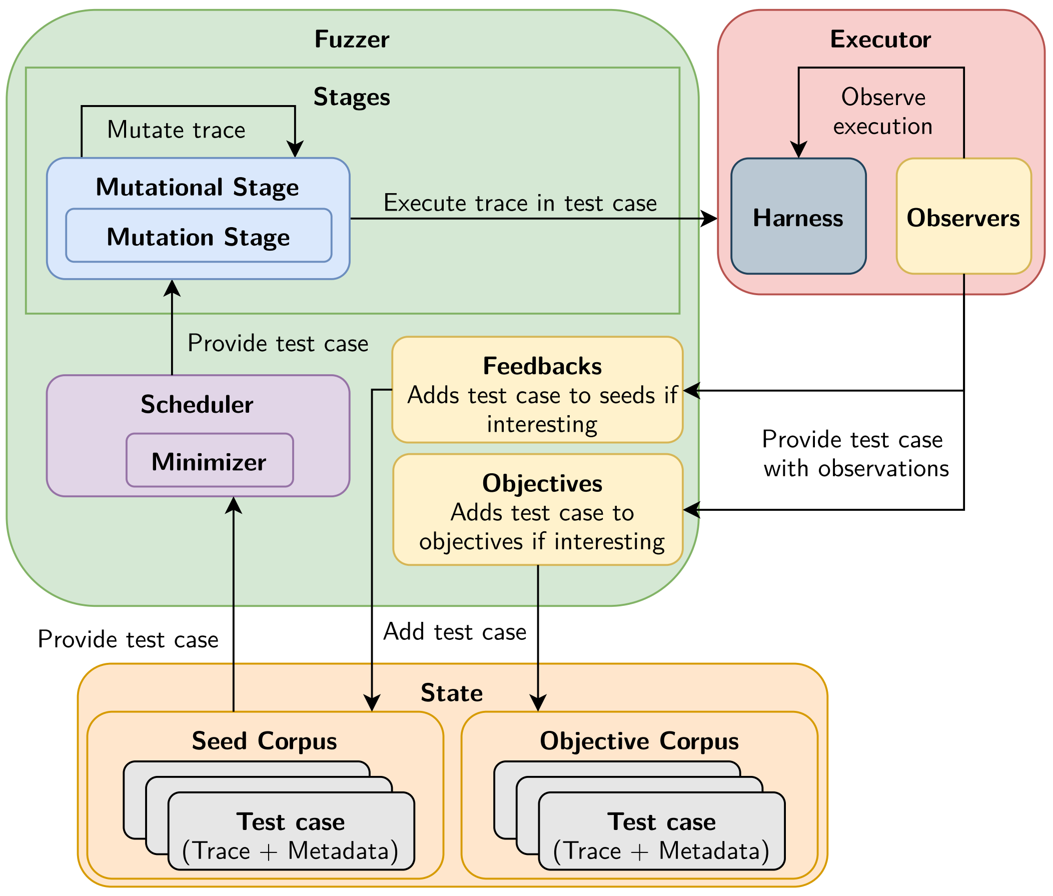


Figure 5.3: This figure shows the fuzzing architecture of tlspuffin by describing the interaction between multiple building blocks. For a better overview the building blocks are separated intro three components, namely Fuzzer, Executor and State. The diagram is read by starting with the Seed Corpus and following the arrows until we either reach again the seed corpus or the objective corpus.

Seed Corpus A corpus is a container for test cases. It can either be stored in memory or on disk. For performance reasons by default tlspuffin is storing test cases in memory. For debugging purposes it can be beneficial to inspect test cases on disk though. Now, the seed corpus is responsible for holding the initial test cases which contain no metadata, as well as future test cases which are discovered throughout the fuzzing campaign. The name for this corpus originates from the initial test cases which are also called seeds and are described in Section 5.3.2.

Scheduler The scheduler is responsible for picking test cases out of the seed corpus. Seed scheduling is an active field of research. This building block is for example the place to implement power schedules which schedules test cases according to an attached energy value [17]. The schedule together with a minimizer also implements input minimization techniques.

Minimizer A minimizer is used within a scheduler to minimize input size. The minimizer uses metadata which is attached after each execution of a test case in order to favor test cases of smaller size which provide the same code edge coverage. The default minimizer which is used in tlspuffin also favors test cases which executed quickly.

Mutational & Mutation Stage After the scheduler and minimizer selected an appropriate test case it is passed to the mutational stage. The same test case is passed to the mutation stage n times. During each iteration the trace within the test case is mutated m times. For each of the m mutations a random mutator is picked. The available mutators are presented in Section 5.3.3. After each iteration the mutated test case is passed to the executor, which executes the fuzzing harness. To sum up, each test case is mutated and executed n times when it reaches the mutational stage. Moreover, $n \cdot m$ mutations are performed.

Executor with Harness The executor calls into the actual library or program. There are multiple ways which depend on the nature of the fuzzing harness. If for example a program is fuzzed which firstly needs to be started using the `execve` system call then a fork-server can be used to accelerate the fuzzing campaign [19]. By using a fork-server we replace the usage of `execve` with calls to `fork` which is much faster. The fuzzing harness of tlspuffin does not require the usage of starting a new process by forking the main one. The fuzzing of TLS libraries can be done in the main process. Therefore, we can use the in-process executor building block. If a crash or timeout occurs, the main process is restarted.

During the execution of a test case we also want to collect information about the run. Observers are responsible for collecting edge coverage data, execution times or information about crashes.

Feedbacks & Objectives Feedbacks and objectives decide whether an execution of a test case is interesting. While the former adds test cases to the seed corpus if it is interesting, the latter adds them to the objective corpus. Different metrics can be selected for each. Both can attach arbitrary metadata to the test case.

New test cases in the seed corpus represent a process in terms of fuzzing. Traditionally, code edge coverage is used to rate a test case as progress. In tlspuffin we are instrumenting the LUT in order to measure edge coverage according to the algorithm first introduced in AFL [77].

New test cases in the objective corpus denote that the test case fulfills the goal specification of the fuzzing campaign. Traditionally, crashes or timeouts are counted as indicators for reaching the goal of the fuzzer. In the case of tlspuffin we let the fuzzing harness crash if a security violation is found. Therefore, no special handling is required in this building block.

Objective Corpus As already mentioned the objective corpus collects all test cases which fulfill the goal of the fuzzing campaign. Like the seed corpus, this can either be stored on disk or in memory. Because we want to persist test cases which led to crashes or timeouts tlspuffin stores this corpus on disk by default.

5.3.1 Multi-Processing Architecture

The fuzzing performance of fuzzers implemented using LibAFL scales linearly with the amount of cores available. The reason for this is that LibAFL employs the message passing pattern to allow communication between multiple worker processes.

A centralized broker process is started which forks itself into worker processes. The parent process is responsible for handling the worker processes as well as coordinating the communication. Each worker process executes the fuzzing harness and therefore the LUT as fast as possible.

In the last section we introduced the concept of seed and objective corpora. Each corpus contains test cases which are interesting according to the definition of the feedback and the objective of the fuzzer. Let us assume that we have many worker processes and one accidentally finds an interesting test case. This test cases should be sent to all other workers, as they benefit from an increase in corpus size. An already interesting test case can probably be mutated further to yield more interesting test cases.

This feature allows us to scale tlspuffin to an arbitrary amount of cores. Especially, in server environments which offer many cores, but lower per-core speed, this is beneficial. A major side effect of this feature is that test cases including the trace definitions need to be serializable to binary data in order to allow communication between the broker and its workers. The solution and implementation for this problem is described in Section 6.5. LibAFL utilizes two ways of communication media. Firstly it bootstraps workers via TCP/IP. After this initial communication shared memory between the process is used for communication. This way of communication allows fast Inter-Process Communication (IPC).

5.3.2 Seeds

In order to demonstrate the capability of the design of tlspuffin and provide a good starting point for the fuzzer, several seed traces have been introduced. Initially, even before the fuzzing starts, a corpus of seed traces is prepared and executed. As soon as the fuzzing starts, one of the seeds in the corpus is selected and mutated.

We crafted several trace definitions by hand in order to provide the fuzzer with a good foundation. By providing a set of seeds we can help the fuzzer to discover other traces which represent different protocol flows. If we would not provide initial seeds, then the tlspuffin would first need to find a trace which does execute without any errors, which could already take a lot of execution. In the following we will go over the main ideas for seeds.

Happy Flow A trace which implements a complete happy flow like depicted in Figure 2.2, forwards messages between two agents. Such traces start with an output step for the agent with the client role. The outputted term is then forwarded to the server agent. This is repeated until both agents claim that the handshake succeeded. The fuzzer can then mutate this trace to find new protocol flows. This resembles a MITM scenario, in which the attacker can eavesdrop and control the public channel.

Example 5.3.1. We are using the trace notation introduced in Section 4.3 to define a trace. The semantic of the used function symbols is given by implementation of tlspuffin. The meaning of each function symbol is not explained further as it is unambiguous. Note that we are using destructors instead of queries in the trace below to make it clear which output is encrypted and therefore not trivially deconstructable in this scenario, because the attacker is a MITM.

$$\begin{aligned} H = &\bar{c}(h_1) \\ &.s(\text{ClientHello}(\text{ProtoVersion}(h_1), \text{Rand}(h_1), \text{SessionID}(h_1), \text{Ciphers}(h_1), \text{Extensions}(h_1))) \\ &.\bar{s}(h_2) \\ &.c(\text{ServerHello}(\text{ProtoVersion}(h_2), \text{Rand}(h_2), \text{SessionID}(h_2), \text{Cipher}(h_2), \text{Extensions}(h_2))) \\ &.\bar{c}(h_3).s(h_3).\bar{s}(h_4).c(h_4).\bar{c}(h_5).s(h_5).\bar{s}(h_6).c(h_6).\bar{c}(h_7).s(h_7) \end{aligned}$$

The Client Hello and Server Hello can be deconstructed because they are sent in plain text. All other messages in the handshake of TLS 1.3 are encrypted and therefore can only be forwarded opaquely. Even though the above example describes the happy flow for TLS 1.3, it is very similar for 1.2.

Client Attacker Additional to the happy flow we also provide a full TLS client encoded in a trace. The details about the client implementation for TLS 1.2 and 13 are provided in Section 6.6. An attack trace only uses one server agent. The client is implemented through the trace definition. Such a trace resembles a scenario in which a client attacks a server, in order to crash the server or mitigate the secrecy, forward secrecy or authentication of other clients. Based on this trace tlspuffin rediscovered the FREAK[7], Heartbleed[31] and CVE-2021-3449 [25] like described in Section 7.2.

Example 5.3.2. Compared to the previous example attack traces are relatively short, but deep. Instead of forwarding messages they construct messages. A TLS 1.3 client without authentication only sends two messages for a successful handshake, namely a Client Hello and a finished message.

$$\begin{aligned} A = {} & s(\text{ClientHello}(\text{ProtocolVersion}, \text{Random}, \text{SessionID}, \text{Ciphers}), \text{Extensions})) \\ & .\bar{s}(h_1) \\ & .s(\text{ApplicationData}(\text{encrypt}(h_1, \text{Finished}(\text{VerifyData}(h_1))))) \end{aligned}$$

The above trace is simplified and does not include the transcript or key shares.

Session Resumption The session resumption feature introduced in Section 2.3 deserves special attention because it requires an already completed handshake because before a session can be resumed. In order to resume a session we firstly have to execute the happy flow trace A and then execute another trace R. The second trace needs to bind itself cryptographically to the initial session.

Example 5.3.3. Let us build now on the previous client attacker trace and continue the trace by resuming the session after it has ended. For this purpose we define a resumption trace which references knowledge from the trace A.

$$\begin{aligned} R = {} & s(\text{ClientHello}(\text{ProtocolVersion}, \text{Random}, \text{SessionID}, \text{Ciphers}), \text{Extensions})) \\ & .\bar{s}(h_2) \\ & .s(\text{ApplicationData}(\text{encrypt}(\text{bind}(h_1, h_2), \text{Finished}(\text{VerifyData}(\text{bind}(h_1, h_2)))))) \end{aligned}$$

Like the trace A the above trace is simplified and does not include the transcript or key shares. We firstly need to execute the trace A and then the trace R using the same LUT instance. The bind function symbol is used to denote a bound between the initial session created in A and the second one. Because there are two different modes `PSK_KE` and `PSK_DHE_KE` for session resumption, tlspuffin contains two different kinds of traces for session resumption. While the first one does not require a fresh Diffie-Hellman key exchange, the latter does. Because session resumption is a new feature in TLS 1.3, there is no equivalent implementation for 1.2.

5.3.3 Mutations

Mutations are an essential concept in feedback driven fuzzing like already mentioned in Section 3.2. A mutator is a function which maps a trace to a new trace by applying mutations. Mutations are responsible for generating new inputs which trigger security violations like those introduced in Section 2.4. The design of suitable mutations is a contribution of this thesis. The process of designing mutations was guided by looking at previous attacks against TLS. By this inspiration we were able to create traces which trigger security violations like demonstrated in Section 7.2. This way we can gain confidence that the fuzzer is able to rediscover vulnerabilities and maybe also new vulnerabilities.

Step Mutations Beurdouche et al. introduced the three mutations **Skip**, **Hop** and **Repeat** [7]. Even though the author did not use these mutations for fuzzing, they represent a good source of inspiration as multiple bugs in the state machines of TLS implementations have been discovered by applying them to messages sent between agents.

The first mutation skips a message in execution of the TLS protocol. This could lead to authentication violations if for example the implementation allows a Certificate Verify message to be skipped. The **Hop** combines two traces by taking a prefix from the first one and continuing with the second trace if it starts with the same prefix. **Repeat** injects messages into a trace by repeating previously sent ones. Note that all these mutations work on a message level and do not mutate of messages. Based on these ideas, we implement the following mutations, which mutate steps. We call them step mutations.

Skip Removes a random input step from a trace. This mutation decreases the amount of messages being sent to the LUT. An example result of this mutation is the skipping of the Certificate Verify message, which would have been sent by a client to perform client authentication. This could present a security violation if the server assumes that the client owns the corresponding private key of the previously sent certificate. In fact this security vulnerability was available in OpenSSL 1.0.0p and 1.0.1k and is called SKIP [7].

Repeat Repeats a random input step which is already part of the trace. This mutation increases the amount of messages being sent to the LUT. A random step in the trace is copied and inserted at a random new position. The repetition of steps could lead to security violations if they are not handled correctly by a LUT as the specification of TLS does not allow sending two messages of the same type consecutively in most cases.

Inject Injects an input step which is not part of the trace. In TLS 1.2 the Change Cipher Spec message is used to tell a peer that the next messages will be encrypted. The communication mode is being changed from plain text communication to encrypted communication. This mutation allows injecting a new message by creating a new input step and generating a random term.

The **Inject** mutations together with the **Repeat** mutation are the only two which can increase the length of a trace. **Skip** is the only mutation which decreases the length of a trace.

Note that the result of the **Inject** mutation can also be achieved by first applying the **Repeat** mutation and then applying a mutation which generates a recipe term in the repeated step. This could be done using the **Generate** mutation which will be introduced later in this section. As we will see below the same redundancy is true for multiple mutators. Redundancy in mutators is a common pattern which is also present in the havoc mutators of bit-level fuzzers like AFL++ [33]. For example decrementing a byte by a value x can be the same as decrementing a byte by $-x$.

Recipe Mutations A major advantage of the fuzzing approach of tlspuffin is that we can mutate terms which correspond to the deep structure of TLS messages. This means that the following mutations are able for example to replace, remove or add extensions. These mutators extend the set of state-of-the-art mutators. By adding the following mutations we are able to change fields within TLS message in contrast of just skipping and injecting whole messages like in the work by Beurdouche et al. [7]. The following mutations are intended to mutate recipes of input steps, and therefore are called recipe mutations.

Remove-and-Lift Chooses a random subterm and replaces it with a random child. This mutation is especially inspired by the idea of removing TLS extensions from a term. A list of extensions can be represented by a term like $\text{append}(\text{append}(\text{new\_extension\_list}, \text{ext}_1), \text{ext}_2)$. By applying the **Remove-and-Lift** mutation it is possible to remove either ext_1 or ext_2 by lifting new_extension_list constant or the second append function.

Replace-Match Replaces a function symbol f with a different one f' where $f, f' \in \mathcal{F}$. This can be seen as changing the implementation of a concrete function. This mutation should only be applied if f and f' have the same arity. Else we would need to fill in missing arguments or remove arguments. The rationale behind this mutation is that we want to be able to change for example constants. Let us suppose there are constants for natural numbers: $1/0, 2/0, 3/0, \ldots \in \mathcal{F}$. As all constants have an arity of 0, it is possible to change a 3 to a zero. Another example is to switch between different hash functions.

Replace-Reuse Replaces a random subterm with a different subterm which is already part of the trace. The term which is being replaced and the replacement term can be part of any recipe in the trace. The rationale behind this mutation is to replay subterms in other parts of the trace. The secure session renegotiation feature of TLS 1.2 which includes the vulnerability CVE-2021-3449 [25] works by transmitting a second Client Hello after the initial handshake [73]. In order to allow mutators to create this trace it is beneficial to be able to copy an already sent message to another input step.

Swap Randomly chooses two distinct subterms and swaps them. While swapping the two terms $t, t' \in \mathcal{T}(\mathcal{F}, N_{pub} \cup \mathcal{H})$ both first get copied and the placed at the corresponding positions in the term. If $t \in \text{st}(t')$ or $t' \in \text{st}(t)$, then it is possible that this mutation becomes a **Replace-Reuse** mutation. If the subterm t' overwrites t, then it is no longer possible to overwrite t' with t, because the original position of t' in the term is gone.

Generate Generates a new and previously unseen closed term from $\mathcal{F}$ and places it at a random location. It is beneficial if it is possible to generate for each function symbol a term which contains it. That way each function $f \in \mathcal{F}$ has the chance to be placed in a recipe.

Query Mutations Traces which represent protocol executions of the TLS protocol require knowledge which has been gathered previously as described in Section 5.1.3. Therefore, the seeds which declare successful executions of the protocol already reference knowledge using queries. In order to use knowledge which is available but not used in the seed traces we require mutations which make is possible to query this. A second reason why we should mutate queries is that they become invalid if other parts of a trace are mutated. For example by placing a list of cipher suites which the server does not support in the recipe of the Client Hello message, the server will respond with a Hello Retry Request message instead of a Server Hello message. The queries which reference fields of the Server Hello message will fail. This means local mutations can invalidate parts or even the whole trace. The following mutations only mutate queries in terms and therefore are called query mutations.

New-Query Replaces a random term with a new query which references knowledge. This mutation allows the introduction of queries by replacing subterms. The query which defines how the variable hole should be filled can be initialized randomly except for the programming language type. Unless the query is used as the new root of the term the type restrictions of the term apply. This means that if a query is set as argument of the function, then the query should obey the argument types of the function. This represents the only way to add variables to a trace if it does not yet contain any. If there are already variables then the **Replace-Reuse** mutation is able to replace subterms with already existing variables.

Mutate-Type Changes the type of queries in root position of recipe term. If the whole recipe term consists of a single query and no function symbols, then we can change its type. If a query is used within a recipe then the type is actually prescribed by its usage. For example if a query is used as a function argument, then the function already dictates which type the query should match. If it is used in the root position of a recipe then we can change the programming language type of the query.

Mutate-Message-Type Changes the message type from which the knowledge should originate. The rationale behind this mutation is that it should be possible to query knowledge from arbitrary messages. By applying this mutation together with the **Mutate-Agent** mutation it is possible to change a query in the following way: Instead of querying the session ID from the Client Hello, query it from the Server Hello.

Mutate-Index Changes the index of the queried message to a random number. Often this is set to 0 is the other parameters of the query already uniquely identify knowledge. If this is not the case then the index is used to select a single one. This is especially true for encrypted messages which look the very same as long as they are encrypted. The idea behind this query is to be able to be able to query and decrypt different messages. Note that it is beneficial to limit the domain of the randomly selected number to $[0, m[$ where m can be set to the maximum amount of knowledge which has been observed while executing the initial seeds.

Mutate-Agent Changes the agent to a different one. The intention behind this mutation is that we want to be able to change which from which agent we want to query knowledge if there are more than one available. This is also very interesting if this mutation is used alongside the **New-Agent** mutation. Together this mutation it is possible to query knowledge from a previously unseen and new agent.

Agent Mutations The above mutators are able to mutate every part of a trace except the definition of agents. Therefore, we will introduce the following agent mutations.

New-Agent Adds a new agent with either a server or client role. Together with the **Mutate-Agent** mutation and other previously defined mutations it is possible to allow mutators to create and generate a complete trace by starting with a completely empty one which has no agents and no steps.

Config-Agent Changes the configuration of already defined agents. As defined in Section 5.1.1 agents require a configuration which can include for example the maximum allowed TLS version. It could even be possible to change the role of an agent from server to client or vice versa. The idea of this mutation is to be able to change this configuration in order to test the LUT in different conditions.

The above agent mutations are not able to remove agents as this would make the whole trace invalid. The reason for this is that steps and queries contain names of supposedly existing agents. The removal of agents would make these references invalid.

Randomness A Pseudorandom Number Generator (PRNG) should be used to generate randomness when selecting terms in the above mutations. That way it is possible to replay mutations by providing the same seed twice and write unit tests. We implemented several tests which test mutations end-to-end by observing whether the intended results are reachable by repeatedly executing mutators. It is noteworthy, that each execution of a trace yields an identical run, under the assumption that the LUT is deterministic. The main source of randomness during fuzzing is the PRNG which is used while mutating traces. This is a very important feature of traces as it allows the reproduction of found vulnerabilities by re-executing them after the fuzzing campaign has stopped. The use of a PRNG also has advantages when testing mutations like we will see in Section 6.9. Deterministic behavior is very important during automated software tests, because results should be reproducible. This can be achieved by seeding PRNG with static values.

Mutation Failures The application of a mutation is a random process and therefore also can fail. For example in the case of the **Replace-Match** mutation an alternative implementation has to be available. Therefore, a mutation can be successful or can fail and is therefore skipped. This means that if a mutation fails, then the trace is not modified. There is a special case in which mutations can also be skipped. If the trace length or a recipe size is becoming too big or small, then the mutation is skipped. The reason this is that we do not want to create infinitely big traces. There are sane limits for trace lengths and recipe sizes which can determined by observing how big the defined seeds are.

Programming Language Types It is noteworthy that the above mutations should respect the types of the programming language in which they are implemented. We already mentioned that reflection is used to compare types in queries. If we perform mutations build terms with inconsistent types then it is guaranteed that the evaluation of the term fails. Statically typed languages have to be very precise here as the execution of concrete functions using reflection will definitely fail if the type of one of the arguments is changed. Even dynamically typed languages can not perform arbitrary operations on arbitrary data. They offer greater flexibility though. With these languages it could be worth to explore whether it makes sense that the mutators ignore types. This introduces the possibility of more failure during evaluation, but also reduces the chance of failing mutators and increases the amount of terms which are possible to construct.

Scope The reference implementation of tlspuffin which is described in detail in the next chapter does not implement all the above mutations. It does implement the most important mutations, namely: **Skip**, **Repeat**, **Remove-and-Lift**, **Replace-Match**, **Replace-Reuse**, **Swap**, **Generate**. With these mutations it is possible to trigger the vulnerabilities mentioned in Section 7.2. Therefore, they represent a subset which is able to demonstrate the capabilities of tlspuffin with room for improvement. The reason for this were time constraints during the development of the implementation. The next important mutators are the query mutators which should be implemented and evaluated in future work.

Categorization In order to give some more intuition on the behavior of the mutations we want to categorize some mutations. The recipe and query mutations can be categorized into 4 categories. Mutations either keep the same term size or increasing or decreasing it. Moreover, mutations can introduce previously unseen terms by generating new ones. The **Replace-Match** mutation never changes the term size as it only changes the implementation of functions. The same is true for the **Mutate-Type**, **Mutate-Message-Type**, **Mutate-Index** and **Mutate-Agent** mutations as they only change already existing queries. The **Remove-and-Lift** mutation can only decrease the term size under the assumption that it did not fail. The **New-Query** mutation keeps the term size the same, unless an inner subterm of the term is replaced. **Replace-Reuse** and **Swap** mutations have the potential to change the size of a term arbitrarily. A special mutation is the **Generate** as it has the potential to introduce previously unseen terms by generating terms from the set of available functions.

5.4 Optimizations

We already highlighted several problems throughout this chapter. The mentioned difficulties are rooted in the design of tlspuffin and can not be solved by an elegant implementation of the design. This section highlights some optimization opportunities which have been discovered while design the fuzzer. It is noteworthy, that the major optimization opportunities are rooted in the design of the fuzzer and also proposes partial or full solutions.

5.4.1 Errors and Robustness

When fuzzing a protocol, we are interested in executions which lead to a successful handshake up until a bug or security violation oracle is triggered. If such an oracle thinks that a trace is interesting, then either a security violation happened or the LUT crashed because of memory management issues. Starting from successfully executing seeds like those introduced in Section 5.3.2, mutations are applied to find more interesting traces. There are multiple reasons for failures which do not indicate executions of interest.

1. A recipe in a trace failed to evaluate because tlspuffin was unable to find knowledge for specified queries.

2. The LUT aborts gracefully with an alert message and terminates the handshake.

3. tlspuffin failed to extract knowledge from the output of the LUT, if for example the output is not parsable.

4. A concrete implementation of a function symbol used in a recipe failed to execute, if for example decryption of a handle failed.

The design of tlspuffin tries to minimize the above reasons for failure, because they indicate that either the trace definition including recipes is malformed or the LUT rejected the trace as invalid. By minimizing these reasons we increase the robustness of tlspuffin.

In Section 5.1.4 we already introduced queries which minimizes the chance that the first reason of the enumeration leads to a failure. By providing a more dynamic approach of matching knowledge with queries we can reference knowledge more robustly.

Example 5.4.1. This example explains how queries increase the robustness of tlspuffin. In TLS 1.3 it is possible to inject optional Change Cipher Spec messages. Because of compatibility reasons with TLS 1.2, most TLS libraries implementing TLS 1.3 allow Change Cipher Spec messages to be sent, but ignore them.

Let us assume now our Happy Flow trace introduced in Section 5.3.2 does not expect Change Cipher Spec messages, but the LUT with the server role emits them. The naïve approach is to reference knowledge just by the order of discovery and assign indices. If the LUT now sends an unexpected additional message, which can safely be ignored the execution of the trace fails.

With the queries of Section 5.1.4 we can use the TLS message type to differentiate between knowledge based on the type of the message from which it originates. If none of the queries in each of the recipes in the trace matches knowledge from a Change Cipher Spec message then it is safely ignored.

This way the trace executes on a LUT which emits Change Cipher Spec message and on LUT which do not emit these messages.

For the other elements in the above enumeration tlspuffin does not offer a method to avoid failures and increase robustness. We rely on a series of mutations which yield traces and trigger interesting behaviors of the LUT. The chance of mutating a meaningful trace without triggering one of the above failures is low. Nevertheless, by repeated execution and mutation eventually new interesting traces can be found.

5.4.2 Recipe Size Explosion

In Chapter 2 we introduced transcript hashes. These hashes can lead to an explosion in term size when modeled through a recipe term. Let us recall that a transcript hash consists of a hash of plain text and decrypted TLS messages like defined in Equation (2.1) in Chapter 2. In order to explain why, we need to highlight two usages of transcript hashes.

- The transcript is used in encryption and decryption of the record layer.
- The transcript is used in the Finished message to cryptographically verify the integrity of the handshake.

In Section 5.3.2 we implement a trace which resembles an attacking client. This client also needs to send the Finished message at the very end of the handshake. The transcript hash which is included in the Finished message depends on the concatenation of the plain text bitstrings of exchanged messages. Usually the series of plain text messages is: 1) Client Hello, 2) Server Hello, 3) Encrypted Extensions, 4) Certificate, 5) Certificate verify and 6) server Finished. The specification of TLS 1.3 explains some cases in which the series of messages deviates from the just mentioned enumeration [60]. In order to build the transcript hash, we need to decrypt the messages sent by the server. Each decryption of a message requires another transcript hash, which includes the plain text of previously exchanged messages.

We can see that the size of a recipe which includes a transcript hash is growing exponentially with the amount of encrypted messages sent by the peer. The size of recipe terms can quickly grow to thousands or function symbols and handles. This is especially true with traces which define session resumption, because they are event longer. There are two solution to this problem.

Remove Transcript Hashes We can patch the LUT to accept arbitrary transcript hashes. To put it in another way, we can set the transcript hash to a bitstring which consists only of zeroes. A trace which defines a client then can use the constant bitstring of zeroes in encryption, decryption and Finished messages[2].

The downside of this approach is that it allows trivial attacks in the MITM. An adversary can for example simply change the cipher suite in the Client Hello and Server Hello messages without any agent noticing.

Persist Transcript Hashes The second solution extracts the computed transcript hashes by the LUT in order to make them available to recipes. By keeping computed transcript in a state, the stateless trace definitions can retrieve the hashes and replace the big transcript terms with a single variable.

This hole in the term is then filled during evaluation of the recipe. We can reuse queries here by simply requesting knowledge by a specific agent and programming language type. As soon as we encounter such a query during evaluation we can inject the extracted transcript hashes.

[2]We implemented zeroing the transcript hashes in OpenSSL 1.1.1k experimentally. See commits `https://github.com/tlspuffin/openssl/commit/f1eeab`, `https://github.com/tlspuffin/openssl/commit/3d11b1` and `https://github.com/tlspuffin/openssl/commit/f5e6dc7`.

The count of distinct transcript hashes in a handshake is finite. In fact there are only the following hashes used: 1) Client Hello..Client Hello 2) Client Hello..Server Hello 3) Client Hello..server Finished 4) Client Hello..client Finished 5) partial Client Hello..partial Client Hello. The partial Client Hello is required for session resumption. Therefore, the effort to instrumentation the LUT is bounded by the number of transcript hashes to extract. The instrumentation for security violation oracles like it will be described in Section 6.7.2 can be reused to extract transcript hashes.

We do not conduct a proof in this thesis to evaluate whether making the transcript available to adversaries weakens the security of the TLS protocol. Therefore, we accept potential false positive results of the security violation oracle. Despite this imminent downside of potential false positives, we include the second approach in the design of tlspuffin in order to take countermeasures against the recipe size explosion. The next chapter highlights major decisions in the implementation.

6 Implementing a Symbolic-Model-Guided Fuzzer

A major contribution of this thesis is the implementation of the fuzzing technique which was discussed in the previous chapter. The goal of the implementation of tlspuffin is to evaluate the feasibility of the approach in terms of effort and outcome. It also reduces the effort for future fuzzing of cryptographic protocols. While fuzzing cryptographic protocols on a higher level than bit arrays sounds promising, it also comes with increased effort as parts of the protocol specification have to be implemented. Furthermore, we want to evaluate which technologies and tools improve both development efficiency and quality.

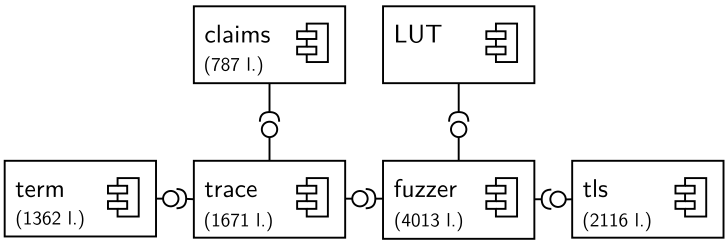


Figure 6.1: Components of tlspuffin. The *fuzzer* module is the core module which uses the interfaces of the *trace* and *tls* modules. The *term* and *claims* modules provide interfaces which are used by the implementation of traces. The LUT is used by the *fuzzer* module as it is responsible for the execution of the LUT and the integration into a fuzzing framework. The number in each component shows the Lines of Code (LOC) it took to implement the module.

The chapter starts with describing early decisions in the development process like the selection of a programming language in Section 6.1. The chapter covers implementation details about the components of tlspuffin as depicted in Figure 6.1. We will investigate which LUTs are available and which one we are targeting in this thesis in Section 6.2. We discuss the module which implements traces and terms in Section 6.3 and Section 6.4 respectively. The implementation of a functional TLS client us described in Section 6.6. Section 6.7 introduces various instrumentations, including a detection of security violations via claims in Section 6.7.2. In Section 6.8 we introduce which data is profiled and how statistics are created. Finally, in Section 6.9 we discuss which practices were applied in the development of tlspuffin and in Section 6.10 we provide a documentation for its command-line interface.

The implementation of tlspuffin consists of over 750 commits and multiple Git repositories. Multiple repositories are required in order to pin exact versions of the libraries which are being tested by tlspuffin. In the next section we will discuss which libraries we actually want to test. All repositories are available at:

`https://github.com/tlspuffin`

6.1 Ecosystem and Programming Language

One of the first implementation decisions is the selection of a programming language. This choice was made in respect to the LUT and in alignment with the fuzzing approach (see Section 6.2 and Section 5.3 respectively). The Rust programming language [48] represents an ideal fit. It allows a simple and standard conform integration of the LUT with tlspuffin, because both C and Rust compile to platform specific machine code. This means that we are able to call functions which are implemented in C from Rust as the latter implemented calling conventions of C. This is a requirement for high-performance in-process fuzzing as mentioned in Section 5.3, as the LUT and tlspuffin share the same memory space.

Safety While the above statements about Rust are also true for other languages like C++, there are other reasons why Rust is a good choice for the implementation of the fuzzer. Rust's type system is safe and expressive and provides strong guarantees about isolation and memory safety [48]. Rust improves the safety of programs by guaranteeing the absence of buffer overflows, stack overflows and accesses to uninitialized memory. While this is beneficial for any program, in the case of fuzzer development this is especially important. Every crash of tlspuffin could indicate a security violation or a bug in the LUT, yielding a false-positive result.

Ecosystem Additionally, the ecosystem around the Rust programming language offers many libraries which are beneficial to the efficiency of development. We are using the Advanced Fuzzing Library (LibAFL) [46], which offers a framework for building arbitrary fuzzers, ring [6], which provides cryptographic primitives and rustls [12], which provides a partial implementation of the TLS 1.2 and 1.3 specification. This library enables us to avoid implementing TLS protocol primitives like key exchanges or schedules. Furthermore, it also provides a parser for disassembling bitstrings into structured TLS messages like required by Section 5.1.3.

Reproducibility The Rust toolchain includes a package manager called Cargo. We are able to manage all libraries, as well as the LUTs through Cargo. This allows reproducible builds of tlspuffin. This is especially beneficial when evaluating the fuzzer and comparing results. We will discuss this advantage of reproducibility again in Chapter 7.

Documentation Lastly, the language features and build system of Rust allow easy generation of documentation. The generated document contains explanations about high level modules, structs, enumerations, traits, and function. An up-to-date version of the documentation is available at:

`https://tlspuffin.github.io/tlspuffin/tlspuffin`

6.2 Library Under Test

The term LUT is used in fuzzing to describe the library which is being tested. Often this is also referred to as the PUT, if the binary is executable and requires no harness to be run. As implementations of protocols are usually packaged as libraries, we are using the term LUT throughout this thesis (see Section 3.1).

Historically, there are many implementations of the TLS protocol. Each implementation offers a different feature set. Some implementations are more conservative, while others try to implement a lot of features. There are several selection criteria for choosing the first target of tlspuffin. Firstly, the LUT should be used actively in production. There is not a high value in finding security vulnerabilities

	High Value	**In-Process**	**CVE Record**	**Open Source**
OpenSSL	✓	✓	✓	✓
LibreSSL	✓	✓	✓	✓
BoringSSL	✗	✓	✓	✓
GnuTLS	✗	✓	✓	✓
Mbed TLS	✗	✓	✓	✓
Botan	✗	✓	✓	✓
JSSE	✓	✗	✓	✓
SChannel	✓	✓	✓	✗

Table 6.1: Table of potential LUTs. These candidates represent a selection of TLS libraries which could serve as a target for fuzzing. The *High Value* column shows whether the library is used in a lot of applications or even serves as a general purpose TLS library for the whole operating system. Such libraries are usually of high value. The next column presents whether the library can be used for *In-Process* fuzzing which requires that the library exposes a C interface. A long list of *CVE Records* indicates that the library is actively used in industry, while at the same time also shows that it had security issues in the past. In order to provide a lightweight but sufficient evaluation the CVE database was searched[4]. Finally, the last column shows whether the library is *Open Source*.

in experimental or academic implementations of TLS. Secondly, for high performance fuzzing we want to do in-process fuzzing and avoid network communication over TCP/IP. That means we want to be able to link against the LUT, in order to abstract away the operating system's network stack. Thirdly, if the LUT has a long record of security issues with high severity, then we can be confident that there are more issues to discover through fuzzing.

Lastly, it is beneficial if the library we are fuzzing is open source. That way we can modify the source code of the LUT in order to employ grey and white-box fuzzing techniques like described in Section 6.7. Traditionally, checks like checksum validations have been disabled during fuzzing to improve the coverage of the testing approach [47]. This LUT transformation is usually easier to perform if the transformations can be performed on the source code and not the binary itself. Even though, removing check sums can lead to false positive test inputs being added to the objective corpus, in practice it is usually easy to sort these out.

Table 6.1 shows a selection of candidates and whether they fulfill the previously stated criteria. OpenSSL and LibreSSL are the only libraries which fulfill all of them. With its initial release in 1998 OpenSSL spawned multiple forks over the years. In fact LibreSSL and BoringSSL are forks of OpenSSL. Both, forks support an interface which is compatible to OpenSSL[1, 2]. Even GnuTLS implemented an experimental OpenSSL compatibility layer[3]. Therefore, by choosing OpenSSL and the more feature conservative LibreSSL as LUTs we can support both through a single interface. Furthermore, we also keep the option open to fuzzing BoringSSL or GnuTLS.

LUT Linking The OpenSSL interface is defined in a C header file called `ssl.h`. Corresponding definitions are available for the Rust programming language. The OpenSSL and LibreSSL LUTs are statically linked to tlspuffin in order to call the defined functions. While this follows the usual linking mode and allows us to reuse existing build scripts for OpenSSL and LibreSSL, this also has a disadvantage. We are not able to fuzz OpenSSL and LibreSSL using the same binary. With the

[1]LibreSSL Compatibility: `https://github.com/libressl-portable/portable/blob/e55410d/README.md`
[2]BoringSSL Compatibility: `https://boringssl.googlesource.com/boringssl/+/HEAD/PORTING.md`
[3]`https://www.gnutls.org/manual/html_node/Compatibility-with-the-OpenSSL-library.html`

current implementation we are not able to switch the LUT implementation during runtime. This also means that we can not spawn an OpenSSL client and a LibreSSL server when executing a trace. One solution to this problem is to avoid in-process fuzzing and instead use a network interface to send messages between multiple implementations. Another solution would be to dynamically load shared libraries using `dlopen` and `dlsym`. The drawback of loading symbols dynamically is that runtime errors can occur.

A further open point is the version of the library which should be discussed. OpenSSL is by far the most wildly used cryptographic library on the internet. Over 84% of all Alexa 1M domains were using OpenSSL in 2017 [55]. The authors did not present data about which versions of OpenSSL. One reason for this is that the TLS specification does not include the possibility to announce or query the version of the implementation.

Modern LUT Versions Because, choosing a LUT depending on its usage is difficult, we want focus on the support for modern or interesting TLS features. For example OpenSSL 1.1.1 is the first version to support TLS 1.3. In the case of LibreSSL TLS 1.3 support was added only in version 3.3.3 on October 18th 2020 – two years later than in OpenSSL. Older versions of the libraries support only the specification of TLS 1.2 which was obsoleted with the release of the RFC for TLS 1.3. The term “obsoletes” here means that the older specification was in fact replaced [62, Section 12]. It is noteworthy, that even LibreSSL 3.3.3 does not support PSKs or session resumption. This and the fact that LibreSSL released TLS 1.3 later than OpenSSL indicates that LibreSSL is more conservative with integrating new features. The latest versions at the time of writing, are OpenSSL 1.1.1k and LibreSSL 3.3.3. Both are fuzzed by tlspuffin like we will see in Chapter 7.

Legacy LUT Versions The older OpenSSL 1.0.x is an interesting target as the protocol state machine implementation for the TLS protocol was rewritten for 1.1.x with the goal of improving maintenance in 2015[5]. OpenSSL 1.0.x also implements some legacy TLS 1.2 feature which could be an interesting target for fuzzing. It is also of high value as it is still used and represents the last FIPS 140-2 capable release [44]. The FIPS 140-2 standard is used to approve cryptographic modules for usage in governmental or banking environments. Unfortunately, the integration of OpenSSL 1.0.x comes with quirks. For example if we want to support forward secret cipher suites which use ephemeral keys, then we need to implement the callbacks `SSL_CTX_set_tmp_rsa_callback` and `SSL_CTX_set_tmp_dh_callback` and manually generate ephemeral keys. Without, this manual setup ephemeral key exchanges are not supported.

In order to reproduce the vulnerabilities mentioned in Section 2.4, we also fuzz OpenSSL 1.0.1f and 1.0.2u additionally to the modern versions of OpenSSL and LibreSSL mentioned previously. A benefit of supporting multiple LUTs is that we gain confidence that the approach to fuzzing of tlspuffin is robust. If the execution of a trace succeeds for multiple LUTs then we can be confident that the trace definition generalizes well and is not specific to a single implementation of TLS.

6.3 Trace Overview

This section describes the core concept of tlspuffin. Traces represent clear instructions of how to achieve a specific protocol flow. There are traces which perform full handshakes according to the TLS specification. There are also traces which trigger vulnerabilities like FREAK[7], Heartbleed[31] or CVE-2021-3449 [25]. These vulnerabilities can circumvent downgrade protection, leak memory or allow denial-of-service attacks respectively. Traces use recipe terms to represent messages. The

[5]`https://github.com/openssl/openssl/commit/f0de395`

implementation of terms will be discussed in Section 6.4. We will discuss the implementation of a trace like in Listing 6.1 and Listing 6.2 in the following.

```
 1  let recipe: Term = term! {
 2    fn_client_hello(
 3      fn_protocol_version12,
 4      fn_random,
 5      fn_session_id,
 6      fn_cipher_suites,
 7      fn_compressions,
 8      fn_client_extensions
 9    )
10  };
```

Listing 6.1: Term which represents a Client Hello message. The term is written using a custom Domain Specific Language (DSL) which will be introduced in Section 6.4.

```
 1  let client_name = AgentName::new();
 2  let server_name = client::next();
 3
 4  let steps: Vec<Steps> = vec![
 5    OutputAction::new_step(client_name),
 6    InputAction::new_step(
 7      server_name, recipe
 8    )
 9  ];
10
11  let trace: Trace = Trace {
12    prior_traces: vec![],
13    descriptors: vec![
14      AgentDescriptor::
15        new_client(client_name, V1_3),
16      AgentDescriptor::
17        new_server(server_name, V1_3)
18      ],
19    steps
20  };
```

Listing 6.2: Declaration of a trace with a sequence of steps.

Term The Listing 6.1 shows a recipe term which describes a Client Hello message. The definition uses a custom syntax which will be discussed in Section 6.4. The provided example should be easy to understand without detailed explanations about the used grammar. The `term!` macro provides an environment in curly brackets in which we can write terms. The listing creates a Client Hello message from several constants like `fn_protocol_version_12` in line 3 which sets the protocol version field to TLS 1.2 or `fn_cipher_suites` which sets the supported cipher suits in line 6.

Trace In Section 5.1.2 we designed traces which use terms to describe messages. A trace can be constructed decoratively as depicted in Listing 6.2. The individual parts of the declaration are discussed in the next paragraphs which cover the trace steps, agent descriptors and prior traces.

Trace Steps The sequence of steps is defined in from line 4 to 9 in Listing 6.2. In line 5 an output step for the agent with a client role is defined. In line 6 an input step for the server is defined. The input step takes the term defined in Listing 6.1 as recipe. As already discussed in Section 5.1.2, actions drive the internal state machine of the LUT forward. An output action induces a gain of knowledge. An input action, with a corresponding term definition uses knowledge and produces a new message which is the passed to the LUT.

Agent Descriptors Each trace consists of a list of agent descriptors. In line 15 of in Listing 6.2 a client and in line 17 a server `AgentDescriptor` is defined. The name of the client is determined by the line 1. The name of the server in line 2 is determined by choosing a different name than the client by calling the `next` function. These two names can be used in queries to reference agents. Each descriptor holds a blueprint for building an agent and initializing a LUT like OpenSSL. It specifies the name of the agent, the version of maximum protocol version which the agent is allowed to use, as well as whether it should act as a client or server. When the trace is executed, an agent is initialized from a descriptor. Depending on the LUT implementation this process involves generating or loading one or multiple certificates, producing keying material, applying server and client configurations and in initiating Random Number Generators (RNGs).

Prior Traces & Agent Reuse Optionally, we can provide a trace with a list of prior traces in line 12 in Listing 6.2. These traces are executed before the main trace and allow attackers to gain knowledge. Prior traces require the reuse of agents. Like in the trace definition in Listing 6.2, each prior trace also specifies agents. By initializing an `AgentDescriptor` with the `AgentDescriptor:new_reusable_server` constructors it is possible to allow tlspuffin to reuse previously initialized agents which match agents from prior trace runs. tlspuffin will match existing agent instances by comparing roles and used other parameters like the protocol version.

The agent reuse feature is fundamentally important because it enables support for session resumption. As already mentioned in Section 5.3.2 session resumption requires multiple handshakes and therefore the reuse of agents. By reusing agents the basic state of the LUT between the execution of a prior trace and the main trace.

One could think that an easy solution in case of session resumption is to merge the steps from prior handshake trace and the main handshake trace and execute it like it is only a single trace. The issue here is that the agent needs to be prepared for reuse after each handshake. Therefore, in the case of OpenSSL and LibreSSL we are calling the function `SSL_clear` which allows to perform a successive handshake, but keeps the basic state which includes secrets which are valid for the lifetime of an OpenSSL server or client. `SSL_clear` resets the LUT and agent. It does not create a completely new agent. For example the ticket key which is mentioned in Section 5.1.1 is not regenerated.

6.4 Term Implementation

As already mentioned, terms are used to represent messages. This section reviews the implementation of terms. It discusses their implementation in tlspuffin and highlights features of the Rust programming language. While Rust allowed an elegant implementation of a DSL, it complicated the implementation because of high safety guarantees.

6.4.1 Term Syntax

In Section 6.4 we already got an idea how terms are declared in the implementation of tlspuffin. The declaration resembles what we theoretically introduced in the term algebra in Section 4.1. A term consists of functions symbols, constants, and variables. The former are implemented using concrete Rust functions. The latter reference knowledge which has been gathered during the execution of output steps.

We are using the `macro_rules!` feature of Rust[6] to implement a DSL. Its grammar can be reviewed in Figure 6.2. An example usage of the DSL within the `term!` macro can be reviewed in Figure 6.3 This embedded language is in fact parsed and type checked by Rust at compile time [37]. The DSL

[6]https://doc.rust-lang.org/reference/macros.html

⟨term⟩	::=	⟨function⟩ | ⟨constant⟩ | ⟨query⟩
⟨function⟩	::=	⟨function name⟩ `(` ⟨argument list⟩ `)`
⟨argument list⟩	::=	⟨argument⟩ `,` ⟨argument list⟩
⟨argument⟩	::=	⟨wrapped function or query⟩ | ⟨constant⟩
⟨wrapped function or query⟩	::=	`(`⟨function⟩`)` | `(`⟨query⟩`)`
⟨constant⟩	::=	⟨function name⟩
⟨query⟩	::=	`(` ⟨agent name⟩`,` ⟨counter⟩ `)`
		⟨opt tls message type⟩ ⟨opt Rust type⟩
⟨opt tls message type⟩	::=	⟨tls message type⟩ | λ
⟨opt Rust type⟩	::=	⟨rust type⟩ | λ
⟨function name⟩	::=	*Rust function reference*
⟨agent name⟩	::=	*Rust integer: name of an agent*
⟨counter⟩	::=	*Rust integer: index of knowledge*
⟨tls message type⟩	::=	`[` *Rust expression for matching TLS messages* `]`
⟨rust type⟩	::=	`/` *Rust type*

Figure 6.2: Grammar of the term DSL in Backus-Naur Form (BNF), in which λ represents the empty string. This language allows declaring terms like `f(g, (h((c, 0)[None]/Random)))`, which first applies the function `h` to a variable of Rust type `Random` with any message type and then applies `f` to the constant `g` and the result of `h`. More details about the variable syntax and its capability to query knowledge can be found in Section 5.1.4. Note that the additional round brackets around functions and variables in the argument list of functions are required to simplify the implementation of the DSL in Rust. The above language is implemented using Rust language features. Therefore, we can make use of the Rust Type system and use a *Rust integer*, *Rust function reference*, *Rust type* and *Rust expression* in the last rules of the grammar.

```
 1  let server_hello: Term = term! {
 2    fn_server_hello(
 3      ((server, 0)[Some(Handshake(Some(ServerHello)))]/ProtocolVersion),
 4      ((server, 0)[Some(Handshake(Some(ServerHello)))]/Random),
 5      ((server, 0)[Some(Handshake(Some(ServerHello)))]/SessionID),
 6      (fn_rsa_cipher_suite12),
 7      ((server, 0)[Some(Handshake(Some(ServerHello)))]/Compression),
 8      ((server, 0)[Some(Handshake(Some(ServerHello)))]/Vec<ServerExtension>)
 9    )
10  }
```

Figure 6.3: An example term which uses the grammar from Figure 6.2. This term can be used in a happy flow trace (see Section 5.3.2) to create a Server Hello message based on knowledge already received from the server agent. The DSL allows the construction of a Server Hello message by referencing learned knowledge like a `ProtocolVersion` or `SessionID` using queries in lines 3, 4, 5, 7 and 8. The message term uses a constant cipher suite in line 6 by using the function symbol `fn_rsa_cipher_suite12`.

allows the declaration of terms in a concise and implementation independent way. By introducing this language layer it is possible to swap the implementation of terms without the need of rewriting the term declarations. This means that by designing this simple DSL we actually introduced a clear interface which separates concerns between the implementation and usage of terms [28].

6.4.2 Knowledge Query Syntax

We discussed in Section 5.1.4 how knowledge is gathered and how we can refer to it using the parameters 1) agent name, 2) TLS message type, 3) Rust type, and 4) index. The grammar in Figure 6.2 specifies how queries can be written. The entry point for the syntax definition of queries can be found in the grammar in 6.2.

Example 6.4.1. Let us suppose that we want to query the server extensions from the Server Hello message. We can write the following term which includes the requested query. We put the query already in some context to make it clearer where data like the server name is coming from.

```
let server = AgentName::new();
// ...

let server_extensions: Term = term! {
  (server, 0)[Some(Handshake(Some(ServerHello)))]/Vec<ServerExtension>
}
```

Rust as Programming Language Conceptually, all the four mentioned query parameters are identical to the parameters mentioned in the design chapter in Section 5.1.4. The only exception are the programming language types. Because we are using Rust as the programming language to implement tlspuffin the programming language types are in fact Rust types. This type of data can directly be derived from the knowledge data via some lightweight reflection in Rust. Rust assigns during compilation every type an identifier which allows equality comparisons. This identifier can be used to compare types for equality during runtime. We can also get the name of functions which is used for example in the serialization and deserialization of traces like described in Section 6.5.

We specify that we want to match knowledge from the agent with the name `server`, which are TLS handshake messages of the kind Server Hello. Moreover, we are saying that we only want knowledge of the type `Vec<ServerExtension>`. If we evaluate the term `server_extensions` in an appropriate context which already learned from a Server Hello message, then we can gather all server extensions.

Optionality of Rust Type If we specify queries in context of another function then we can easily derive the Rust Type automatically without specifying it explicitly. The reason for this is that queries in the argument positions of a function inherit the declared argument types. By allowing them to be optional we can simplify the delcaration of queries.

Example 6.4.2. In the following term we can trivially omit the Rust type, because it is already obvious from the function signature of `fn_remove_first_extension`. This function takes a list of server extensions and returns a list of server extensions.

```
let server_extensions: Term = term! {
  fn_remove_first_extension((
    (server, 0)[Some(Handshake(Some(ServerHello)))]
  ))
}
```

If we use a query in the root position of a term like in Example 6.4.1, then we are required to specify it.

6.4.3 Term Data Structure

The implementation of terms requires a tree-like data structure which allows modifications through the mutations as specified in Section 5.3.3. Essentially, we want to be able to replace subterms like they are defined in Section 4.1 with arbitrary other subterms. We also want to be able to manipulate subterms by replacing their function symbol with a different one.

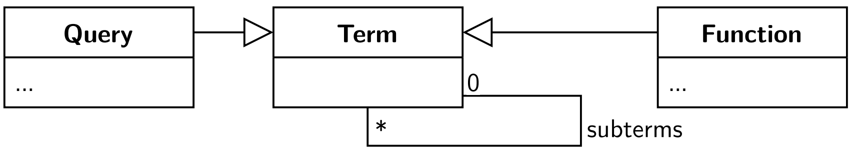


Figure 6.4: UML diagram of a naïve term implementation. A term is either a variable which uses a query to reference knowledge or be a function. For simplicity additional attributes have been omitted from the diagram. A term contains zero or more subterms. Subterms are allocated on the heap and not on the stack.

The naïve implementation of a term is shown in Figure 6.4. The figure shows that a term is either a variable which uses a query to reference knowledge or be a function. It also shows that a directed acyclic graph with the limitation that each child can have only one parent node, which corresponds to trees. In the following we will use the terms node and tree for terms and subterms interchangeably. The direct subterms of a term are also called children of a term. The following paragraph explains an operation which is required to implement the term mutations firstly mentioned in Section 5.3.3.

Mutations To implement mutations, we want to be able to pick and reference a random subterm mutably from all tree notes which match a specific condition. The following example describes an explicit use case of the just mentioned operation.

Example 6.4.3. To explain the algorithm for a sample mutation more concretely, let us start with an example. The **Replace-Match** mutation replaces the function symbol in a random node of the term with a randomly chosen different one.

Let us define $\mathcal{F} = \{f, g, h\}$, $\mathcal{H} = \{h_1, h_2\}$ and $t = f(g(h_1))$. Even though the function and handles in the formal model do not have types, the function symbols in the implementation of tlspuffin do have types. This means we only want to perform mutations which respect types. Else the evaluation will definitely fail. We assume that the argument types and return types of the function symbol g and h are equal.

Firstly, we select a random function symbol from $\mathcal{F}$ without any restriction. Let us suppose now that we selected the function symbol h. Now we select a random subterm s of t, which matches the following condition: The term t must be a function and the argument and return types must match those of h. A promising and compatible candidate is $s = g(h_1)$. After finding a matching term s, we replace the function symbol g with h resulting in a new term $t' = f(h(h_1))$. We successfully replaced a random subterm with another random subterm.

Aliasing XOR Mutability Problem A simple way of randomly selecting a subterm is to map the term tree into a list, filter it according to a condition and then select a random subterm uniformly. While flattening the tree into a list, we have at some point more than one mutable reference to nodes. This violates the design principle *aliasing XOR mutability* of Rust. The *aliasing XOR mutability* feature prevents data races by allowing only a single mutable reference or multiple non-mutable references.

Let us assume we start to unwind the tree by adding a mutable reference to the root node to the list. Now we iterate mutably through the children of the root node and add these references to the list. Even with this simple example of adding nodes up to a depth of two shows that we would have two mutable references to the root node. Once we mutably reference the node itself in the list, and once we reference the children. The interested reader can refer to [76] for a more detailed discussion about tree, graphs and linked lists and solutions to mutability challenges. It explains a new concept which can be used to implement these data structures efficiently and safely.

Solution using Reservoir Sampling In tlspuffin we first find a random node non-mutably from all tree nodes which match a certain condition. Then we perform a second pass to re-find the selected node mutably. An elegant way to implement this is utilizing the random selecting using reservoir sampling [71]. After adapting this algorithm to trees, we are able to select a random node from a tree of unknown size in a single recursive iteration through the tree. The idea of the algorithm is to create a reservoir in which a single node fits. Additionally, to the node we also always put a path from the root to the node in the tree. This reservoir is getting overwritten with gradually decreasing probability proportionally to the amount of already visited nodes. After recursively iterating once over the tree the reservoir contains a single uniformly randomly selected node and the path to it. In the second pass we can mutably re-find the found node by following the path. The Figure 6.5 shows an implementation using pseudocode which is inspired by Rust.

Alternative Data Structures Other data structures could also be used to implement operations required for mutations more easily. One could use arena allocators to manage memory in a flat list or reference counting to address the *aliasing XOR mutability* restriction. These variants represent more flexible solutions compared to the one described above, but comes with the cost of increased

```
fn reservoir_sample_trace(trace: &Trace) -> Option<&Term> {
  let mut reservoir: Option<&Term> = None;
  let mut visited = 0;

  for step in &trace.steps {
    let mut stack: Vec<&Term> = vec![step.recipe];
    while let Some(term) = stack.pop() {
      for sub_term in &term.subterms {
        stack.push(sub_term);
      }
      visited += 1;
      if random_between(1, visited) == 1 {
        // gradually decreasing probability $\frac{1}{\text{visited}}$ of filling the reservoir
        reservoir = Some(term);
      }
    }
  }
  return reservoir;
}
```

Figure 6.5: Pseudocode for randomly selecting a subterm. The recursive iteration through the tree is implemented iteratively using a stack. The syntax inspired by Rust. `&Term` denotes a reference to some term. `vec![...]` allocates a new vector.

complexity. Arena allocators require complex memory management because terms would be manually allocated instead of using the system allocator. Reference counting opens the possibility to memory leaks in case of cyclic references and reduces performance during runtime [76]. It is noteworthy, that even future changes in the implementation of terms do not require major changes because trees are encapsulated through the grammar of the DSL introduced previously in Section 6.4.1.

6.4.4 Term Visualization

In Section 6.5 we described how we can transform traces into files to make them more accessible. Nonetheless, investigating what a trace actually does, is not yet easily possible for a security researcher. Therefore, tlspuffin offers a textual and a graphical way to visualize terms. This visualization can easily be extended to traces.

Graphical Representation We are utilizing the GraphVIZ library [34] to render a graphical representation and persist it as Scalable Vector Graphics (SVG), Portable Document Format (PDF) or rasterized image like Portable Network Graphics (PNG). The Figure 6.6 shows an example visualization of a Client Hello message term. If the fuzzer finds traces which violate a bug or security violation oracle, then users of tlspuffin can quickly gain an overview what the trace is doing. Deviations from the TLS specification can be vetted and assessed.

Text Representation The just mentioned visualization is beneficial to users of the fuzzer who want to investigate traces. For developers of tlspuffin it can be beneficial not to have a textual representation, which can be outputted on the console. An example can be observed in Figure 6.7. The textual representation resembles the notation we used in the formal model in Section 4.1.

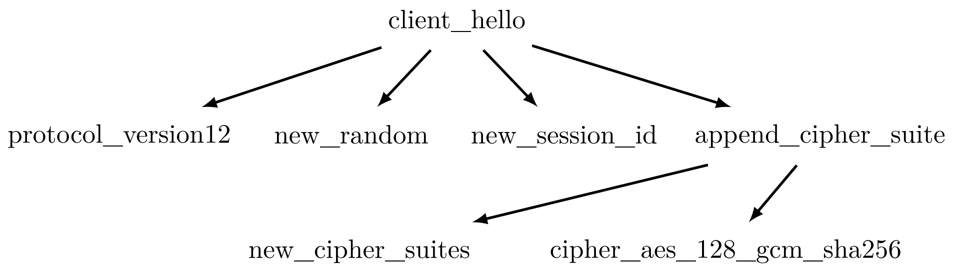


Figure 6.6: Visualization of a simplified Client Hello message term which uses only constants. The client extensions and compressions fields have been removed because of space constraints.

```
fn_client_hello(
        fn_protocol_version12 -> ProtocolVersion,
        fn_new_random -> Random,
        fn_new_session_id -> SessionID,
        fn_append_cipher_suite(
                fn_new_cipher_suites -> Vec<CipherSuite>,
                fn_cipher_suite13_aes_128_gcm_sha256 -> CipherSuite
        ) -> Vec<CipherSuite>
) -> Message
```

Figure 6.7: Textual representation of a simplified Client Hello message term which uses only constants. The client extensions and compressions fields have been removed because of space constraints. The indentation is also visible when outputing the term on a console. Compared to the graphical visualization in Figure 6.6, the textual representation also outputs the Rust types.

6.5 Test Input Serialization

The term serialization (or marshalling) describes the process of converting volatile data structures into a format which can be persisted on disks. Usually, serialization can be understood as a function from in-memory data structures to a string or bitstring. The reverse process is called deserialization (or unmarshalling) and creates in-memory data structures from strings or bitstrings. By first serializing a data structure to a bitstring and then deserializing it we usually obtain a value which is equal to the initial value.

In classical fuzzing campaigns bitstrings are used as test input. This is sufficient for fuzzing for example file formats or command line parameters. Bitstring are naturally serializable by using the identity function for serialization and deserialization. This has three benefits:

- We can store the seed traces on disk and load them during the initialization of the fuzzer. That way we adjust the initial seed corpus my adding or removing files on disk.
- If we find a test input through fuzzing which triggers a bug, then we can rediscover the crash by replaying the test input after the runtime of the fuzzer.
- The serializable format is portable between different computing platforms and can be shared with other security researchers.

As we already discussed in Section 5.1.2, the input space of tlspuffin are traces. We want to lift the three advantages from traditional fuzzing to tlspuffin. We are using the Rust library Serde [69] for serialization. The additional library postcard [52] provides an efficient binary data format. This means that we are not serializing to strings but to bitstrings in tlspuffin. The flexibility of Serde also allows to change the data format to Extensible Markup Language (XML) or JavaScript Object Notation (JSON) without additional effort.

Runtime Linking Generally, the Serde library provides a simple framework for making arbitrary data serializable. The major challenge for tlspuffin is to make references to implementations of concrete functions and data types serializable. The actual program which is executed during the execution of a Rust function can not be serialized directly. This is a usual limitation of serialization as loading code during runtime would require packing code in file which can be loaded by operating systems like a shared library.

Instead, we only persist the names and argument shape of concrete functions during serialization. When we deserialize a bitstring, we perform a lookup and try to link the function name to existing Rust functions. If no such function exists, then the deserialization fails. Therefore, the trace file must match the tlspuffin version. If we remove or rename a concrete function from tlspuffin which is required by a trace file, then the deserialization of the file will fail.

Example 6.5.1. For example the function name `"tlspuffin::tls::fn_protocol_version12"` with the return type `"rustls::msgs::enums::ProtocolVersion"` is linked to the Rust function:

```rust
pub fn fn_protocol_version12() -> Result<ProtocolVersion, FnError> {
  Ok(ProtocolVersion::TLSv1_2)
}
```

6.6 Functional TLS Client

All traces which go beyond just forwarding messages between two agents, require functions symbols. Traces which just forward messages actually can be declared just by queries. Each function symbol requires a concrete implementation, which has to be done by manually writing code in Rust. Therefore, we actually need to implement full TLS clients in servers. For this thesis we only implemented a TLS client trace, which needs to be able to perform a full handshake with a LUT successfully, and also be able to deviate from the specification. The *tls* module of tlspuffin, which was firstly mentioned in at the beginning of this chapter, provides plenty of function symbols which are available to the fuzzer. In this section we want to discuss which function symbols we implemented, guided by the required traced introduced in Section 5.3.2.

TLS Ecosystem in Rust On the one hand, some functions are rather simple, like returning cipher suite identifiers or creating a list of them. On the other hand, the functions which implement key exchanges and schedules are more complex. Implementing a ECDHE key exchange or schedule from ground up is quite an effort in terms of time. The ring [6] Rust library provides implementations of cryptographic primitives and therefore represents a genuine help in implementing key exchanges, key schedules and encryption and decryption of TLS messages. The rustls [12] library complemented the ring library by providing a reference implementation of TLS 1.2 and 1.3. We were able to reuse code of rustls to implement mandatory parts of the TLS 1.2 [61] and 1.3 [60] specification quickly. Because rustls library is targeted for end-users of TLS we had to adjust its source code in order to expose internals.

Trace Development In Section 5.3.2 we introduced traces which should serve as seeds for the fuzzing process. We use these seeds as guideline to implement function symbols. Within 2 person-weeks a full TLS 1.2 and 1.3 handshake was implemented using function symbols. This implementation made it possible to implement the **Attacker Trace** from Section 5.3.2. The implementation of the **Session Resumption Trace** for TLS 1.3 took another half person-week to implement by recycling code from rustls. Figure 6.6 gives an idea on how the recipes of traces which implement a client look like. Each message, field within a message, cipher suite and extension required for a TLS handshake needs to be implemented. In total over 160 concrete implementations for function symbols have been manually programmed using almost 1700 lines of Rust code. There are mostly four categories of function symbols. Table 6.2 shows the composition of the function symbols by category.

Category	#	Description
Constants	19	Consist of selected constants like numerical values like 1, 2, 3, ... The category also contains numbers which represent edge cases like very high numbers.
Messages	27	Provides function symbol implementations which build Client Hello, Server Hello or Finished messages. These functions do not contain a lot of logic, but create structured messages.
Fields	19	These function symbols are used in messages and return simple data like protocol versions or cipher suites, as well as signed hashes like verification data.
Extensions	77	This category is the biggest and implements many extensions which can be sent in Client Hello, Server Hello, Session Tickets or Certificate messages.
Others	20	This category consists of various function symbols which encrypt or decrypt TLS messages and construct transcript hashes.

Table 6.2: The table shows all four categories of functions symbols which have been implemented in tlspuffin. The second column denotes of how many function symbols the category consists. Cryptographic operations are either used directly in the concrete function, or helper functions are used. For example, there is a range of helper functions which perform key derivations according to the key schedule like described in Figure 2.3.

Mandatory Cipher Suites Various resources have been consulted in order to decide which function symbols should be added and implemented. For TLS 1.3 cipher suites we were able to select the mandatory ones from the RFC [60]. Because all cipher suites for TLS 1.3 are forward secret and only differ in the kind of symmetric encryption algorithm and hash size, it is sufficient to implement only a single cipher suite namely `TLS13_AES_128_GCM_SHA256`. The reason for this is that we are fuzzing the logic of the handshake and assume that the cryptographic operations used during encryption and hashing are secure. In fact, the cipher suites of TLS 1.3 only differ in slightly in the choice of key-lengths, hash lengths. The only cipher suite which stands out is `TLS_CHACHA20_POLY1305_SHA256`, which uses ChaCha20 instead of AES.

For TLS 1.2 we have a lot more options which cipher suites. Due to time constraints we also focused on a single cipher suite which is `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`. This cipher suite roughly corresponds to the one we use for TLS 1.3. Additionally, it specifies which algorithm is used for authentication, which is RSA. In TLS 1.3 this algorithm is negotiated through the Signature Algorithms extension and not through the cipher suite definition. Future work on the TLS 1.2 client trace could include to support for other cipher suites which do not support forward secrecy like `TLS_RSA_WITH_AES_256_CBC_SHA` or require a PSK like `TLS_PSK_WITH_RC4_128_SHA`.

Mandatory Extensions Similar to the cipher suites, the RFC of TLS 1.3 [60] also specifies mandatory extensions, which include for example the Supported Versions, Signature Algorithms or Key Share extensions. In order to perform a successful handshake we had to implement all of them expect for the Cookie extension. Additionally, we added non-existing extensions which are unknown to any LUT in order to be able to test their behavior if they receive the fake extension.

Moreover, we also consulted the IANA assignments documents[7] to gain a historical overview of all extensions of all TLS versions. We implemented all which were also available in the TLS reference implementation rustls. Additionally, we also implemented the Heartbeat message and extension in order to rediscover the Heartbleed vulnerability like shown in Section 7.2.

Increasing Feature Coverage In conclusion, we can say that implementing as many function symbols of TLS is beneficial for the fuzzer. Eventually, the fuzzer should be able to trigger every feature of a LUT which is enabled. The task of deciding which features and corresponding function symbols should be implemented is difficult and more research needs to be conducted. A clear method of deciding which features are worth fuzzing and which not is still missing.

By providing a diverse set of function symbols we already aimed to increase the reached states of the LUT. This includes for example function symbols for session resumption and PSKs. Future work, should focus on implementing more functions, and also testing their usage by writing test traces for the implemented features to validate that they are usable. Many of the 160 implemented function symbols in tlspuffin have not been tested in concrete example traces which use the symbols. Alternatively, future work could include to measure what percentage of function symbols have been used in successfully executing traces during fuzzing campaigns. This would resemble some kind of function symbol coverage.

6.7 Instrumentation

The feedback loop in the fuzzing campaigns of tlspuffin is heavily influenced by feedback gathered by instrumenting the LUT. In the fuzzing algorithm mentioned in Section 5.3 we use code coverage information to guide the fuzzer. Moreover, we mentioned in Section 5.2 that we are using another kind of instrumentation to detect security violations. Furthermore, we also want a more fine-grained detection of memory issues by sanitizing memory access. All three kinds of instrumentations are applied by transforming the source code of the LUT. This is possible because we are using open-source LUTs like determined in Section 6.2. In the following we will explain how both instrumentations are implemented and applied to the LUT.

6.7.1 Code Coverage Instrumentation

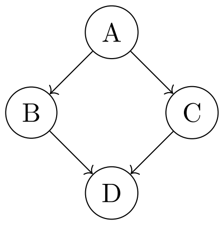
There are compilers for the C/C++ programming language like LLVM which offer an instrumentation which observes which edges in the source code are taken. This is also called the edge coverage. By executing an instrumented program, we can observe which edges and which percentage of edges are taken.

In order to understand what counts as an edge, we first have to introduce Control-Flow Graphs (CFGs). A CFG describes all possible execution paths through a program with a directed graph.

[7]The list can be retrieved in various formats here: `https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml`

Example 6.7.1. Let us consider the following program written in C, which sets the value behind a pointer `a` to the value `b + 1`. Finally, `b` is returned. We can assign each continuous code block a name. Line 2 is block A, line 4 is block B, line 6 is block C, and line 8 is block D. These blocks are also called basic blocks. The CFG for the code example shown right next to it.

```c
int foo(int *a, int b) {
  b = b + 1; // A
  if (a) {
    *a = b; // B
  } else {
    return 0; // C
  }
  return b; // D
}
```


$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B \arrow[dr] & & C \arrow[dl] \\ & D & \end{tikzcd}$$

Usage The `trace-pc-guard` instrumentation of LLVM[8] adds for each basic block a function call `__sanitizer_cov_trace_pc_guard(uint32_t *guard)`. Each value of `*guard` corresponds to an edge. In order to enable this instrumentation we have to compile our LUT with the flag `-fsanitize-coverage=trace-pc-guard`. We only need to add this flag in the compilation of the LUT. The fuzzing harness which is written in Rust does not need to be instrumented. LibAFL implements edge coverage detection using a memory map like described in [77]. With each hit of an edge LibAFL increments a corresponding value in a memory map. Because the amount of edges is finite, it is usually enough to allocate 64 kB in order to count the hits for all edges. At least this value is sufficient for all fuzzed OpenSSL versions and LibreSSL.

Future work could include other instrumentation to measure coverage. Edge coverage might not adequately represent process in fuzzing a protocol. In fact, what we ideally want is path coverage. Each test input which represents new distinct flow through the protocol should be counted as interesting and therefore as progress. Unfortunately, path coverage is impractical to measure because there are potentially infinite paths through a program. A solution could include a domain-specific metric for protocol executions.

6.7.2 Security Violation Oracle Instrumentation

In Section 5.2 we introduced several checks which we want to perform on sequences of claims. These claims contain information from the LUT. Claims in tlspuffin are recorded after a message is sent and received. Alternatively, we could decide to record claims only after each flight of messages. A flight denotes consecutively sent messages until it is the peer's turn. This is not ideal, because a security violation could happen within a flight. If it happens within a flight, then it is maybe not detectable after it finished.

Interface We define an interface through a C header file which needs to be implemented by the LUT. This means we need to instrument the LUT manually. The interface works by registering a closure in the LUT. The following listing provides a declaration of the interface.

```c
typedef enum ClaimType {
    CLAIM_CLIENT_HELLO,
```

[8] `https://clang.llvm.org/docs/SanitizerCoverage.html`

```
 3      ...
 4  } ClaimType;
 5
 6  typedef struct Claim {
 7      ClaimType typ;
 8      ...
 9  } Claim;
10
11  typedef void (*claim_t)(Claim claim, void *ctx);
12
13  void register_claimer(const void *tls_like, claim_t claimer, void *ctx);
14
15  void *deregister_claimer(const void *tls_like);
```

By calling `register_claimer` in line 13 from Rust we can register a closure. In order to free memory we also can de-register it using `deregister_claimer`. The `Claim` struct in line 6 holds all the information which should be exposed. Each time a new message is sent or received the closure is called which makes this struct available in Rust. Each claim also contains a type which signals which message was sent.

Transcript Hashes As already introduced in Section 5.4.2, we are using this interface also for exposing transcript hashes. In this case the `Claim` struct specifies for example the type `CLAIM_TRANSCRIPT_CH` and includes transcripts hash from Client Hello..Client Hello.

Interface Implementation Claims can contain information which is exposed through an API of the LUT. Examples are the OpenSSL functions `SSL_get_tmp_key` and `SSL_get_peer_tmp_key` which return the ephemeral public keys used in a ECDHE key exchange. But not all data is retrievable through an API. For example the used signature algorithm can only be acquired by accessing internal data of OpenSSL. This justifies why we need to implement the instrumentation within the LUT. If we would implement the interface within the LUT, then we would not have access to relevant information. Right now we are exposing the assumed protocol version, the session ID, randoms, certificates, ephemeral keys, signature algorithms, ciphers, transcripts, and the secrets shown in the key schedule for TLS 1.3 in Figure 2.3. We implemented the full claim interface only for OpenSSL 1.1.1k with TLS 1.3. Using the agent claims we implemented policies for the security violation oracle in Rust like described in Section 5.2.

6.7.3 Memory Sanitization

In order to detect bugs with memory access it is beneficial to instrument the LUT with ASAN [65]. ASAN employs source code instrumentation together with a runtime library in order to detect memory management issues. Bugs which can be detected amongst others are use-after-free bugs and heap and stack buffer overflows. The Heartbleed vulnerability for example is caused by a heap buffer over-read.

Usage ASAN comes like the edge coverage detection with the LLVM toolchain. In order to enable it the LUT needs to be compiled with the flag `-fsanitize=address`. Furthermore, we need to instruct the Rust compiler to link against the ASAN runtime. The default of ASAN is to exit the current process if a bug is detected. This represents an incompatible behavior for LibAFL. Instead, we need to configure ASAN to send an `abort` signal to its own process. This can be done by setting the environment variable `ASAN_OPTIONS` to `"abort_on_error=1"`[9].

Performance Memory sanitization usually introduces a significant performance overhead. In average programs experience a slowdown of 73% [65]. Even though we only want to detect memory issues in our LUT and enable sanitization only for the LUT and not for the whole fuzzer, we also experience slowdowns when allocating heap memory from our fuzzing harness written in Rust. The reason for this is that ASAN comes with a runtime library. This library replaces the `malloc` and `free` functions which custom implementations, in order to detect heap-based bugs. The issue is that Rust also uses `malloc` internally. Because the runtime library is loaded globally, the slowdown is perceptible in every part of the fuzzer. A fact to be aware when using ASAN while fuzzing is that it allocates a lof of virtual memory on 64-bit systems[10]. While this memory is not actually used, it is easy to reach limits opposed by the operating system.

6.8 Fuzzer Statistics

We want to create statistics using profiling data. Statistics can provide insights in the behavior of a fuzzer. Together with tlspuffin we provide a toolset which consists of two components. The first one is implemented in tlspuffin and gathers profiling data during a fuzzing campaign. The second one evaluates this data to generate summarized statistics.

Profiling The term profiling refers to analyzing a program during its runtime. Classically, memory allocations or CPU cycles are counted. In tlspuffin we profile the fuzzing campaign by collecting more high-level data per worker client.

Total Executions The total amount of executed traces.

Executions per Second The rolling count of trace executions per second.

Trace Errors Various errors like can happen during the execution of a trace. We are counting these errors per category. The three categories are described in the following. The LUT can raise an alert which terminates the TLS session. Moreover, a term can fail to evaluate, because a query did not find any applicable knowledge. Finally, there can also be errors which happen within concrete implementations of function symbols. For example authenticated encryption can fail if the used key does not match the cipher text.

Seed Corpus Size The size of the seed corpus.

Objective Corpus Size The size of the objective corpus.

Coverage A ratio between discovered code edges and the maximum code edges of the LUT.

Performance Data Information about the CPU cycles spent in various steps of the fuzzing loop. It allows for example to calculate how much time has been spent mutating traces, scheduling traces or executing the actual LUT.

Trace Data Insights about the minimum, maximum, and average trace length and recipe term size.

[9] `https://github.com/google/sanitizers/wiki/AddressSanitizerFlags`
[10] `https://afl-1.readthedocs.io/en/latest/notes_for_asan.html`

The above data is persisted in the JSON format. Because we are writing this profiling data to disk as it is being generated we actually need a streaming JSON format [74]. Essentially, we are writing one JSON object after the other.

Evaluation The Python programming language offers good tooling for data processing. Therefore, we implemented the data evaluation pipeline using an efficient JSON parser, Numpy [36], and Matplotlib [39]. The implementation of the Python module is called `tlspuffin-analyzer` and is included in the Git repository of tlspuffin. The final result of this pipeline are plots which we will investigate in the evaluation in Chapter 7.

6.9 Development Methodology

During the development of tlspuffin various practices have been applied which helped to reach certain quality goals. The paragraphs below describe best practices for software development.

Unit Testing Testing code is an essential practice in order to gain confidence in the correctness of code [53]. We wrote around 45 in tests for tlspuffin which execute whole traces, test serialization or mutations. With this test suite we reach a line coverage of 72%.

The execution tests for traces allowed us to test traces against different LUTs. For example, we run the same traces against LibreSSL 3.3.3, OpenSSL 1.0.1f, OpenSSL 1.02 OpenSSL 1.1.k, depending on the features they support. Obviously, we can not run a TLS 1.3 trace against OpenSSL 1.0.x because the older version of OpenSSL does not support TLS 1.3.

These tests helped numerous times during the development of tlspuffin to detect regressions. Changes in the fuzzing harness might affect only a subset of traces. By being able to executing all traces easily, programming mistakes could be caught easily. The test suite does not only contain traces which reach a state in which a handshake is successful. The suite also contains traces which trigger a vulnerability like the CVE-2021-3449 [25].

Furthermore, unit tests together with the ASAN instrumentation introduced in Section 6.7.3 we can also detect the causes of memory leaks without running the actual fuzzer. ASAN is able to print the exact source of memory leaks after the whole test finished. This tool helped to debug memory leaks which were caused while writing C/C++ code or Rust code which uses the `unsafe` keyword.

Micro Benchmarks Fuzzing is a performance critical task. Research in fuzzing is engaged with increasing the performance of executing the testing harness [75]. The executed iterations per second represent a crucial metric in order to estimate how fast the state space can be explored. Therefore, tlspuffin uses the criterion.rs [38] Rust library to benchmark the execution of traces and evaluation of terms.

Continuous Integration The just mentioned tests and benchmarks can be executed in a Continuous Integration (CI) environment. With each commit and push to the Version Control System (VCS) a new compilation process is started. After successful compilation the tests are run and the documentation of tlspuffin is generated. If everything succeeds, then the documentation is published to the internet. This way we can make sure that tlspuffin can be compiled independent of the environment of developers.

Issues While developing the reference implementation for the design described in this thesis 97 issues have been created, out of which 72 have been resolved. The open-issues document tasks which can be done to improve the implementation of tlspuffin. A kanban board was utilized to prioritize and keep track of development tasks.

6.10 Command-line Interface

The syntax for the command-line of tlspuffin is:

tlspuffin [⟨*options*⟩] [⟨*sub-command*⟩]

where **tlspuffin** is the binary of the fuzzer, ⟨*sub-command*⟩ is an optional sub-command, and ⟨*options*⟩ are global optional command-line parameters which are used in every sub-command. If no sub-command is specified, then the fuzzing procedure is started.

Global Options Before we explain each sub-command, we first go over the options in the following.

-c, –cores ⟨*spec*⟩
This option specifies on which cores the fuzzer should assign its worker processes. It can either be specified as a list by using commas `"0,1,2,7"` or as a range `"0-7"`. By default, it runs just on core 0.

-i, –max-iters ⟨*i*⟩
This option allows to bound the amount of iterations the fuzzer does. If omitted, then infinite iterations are done.

-p, –port ⟨*n*⟩
As specified in Section 5.3.1 the initial communication between the fuzzer broker and workers happens over TCP/IP. Therefore, the broker requires a port allocation. The default port is 1337.

-s, –seed ⟨*n*⟩
Defines an initial seed for the PRNG used for mutations. Note that this does not make the fuzzing deterministic, because of randomness introduced by the multiprocessing (see Section 5.3.1).

Sub-commands Now we will go over the sub-commands **execute**, **plot**, **experiment**, and **seed**.

execute ⟨*input*⟩
This sub-command executes a single trace persisted in a file. The path to the file is provided by the ⟨*input*⟩ argument.

plot ⟨*input*⟩ ⟨$format$⟩ ⟨$output\_prefix$⟩
This sub-command plots the trace stored at ⟨*input*⟩ in the format specified by ⟨$format$⟩. The created graphics are stored at a path provided by ⟨$output\_prefix$⟩. The option –multiple can be provided to create for each step in the trace a separate file. If the option –tree is given, then only a single graphic which contains all steps is produced.

experiment
This sub-command initiates an experiment. Experiments are stored in a directory named *experiments* in the current working directory. An experiment consists of a directory which contains 1. a JSON file which contains statistics, 2. a log file, 3. a set of traces which represent crashes, and 4. a text file which contains metadata of the experiment. The title and description of the experiment can be specified with –title $\langle t \rangle$ and –description $\langle d \rangle$ respectively. Both strings are persisted in the metadata of the experiment, together with the current commit hash of tlspuffin, the LUT version and the current date and time.

seed
This sub-command serializes the default seed corpus in a directory named *corpus* in the current working directory. The default corpus is defined in the source code of tlspuffin using the trace DSL.

Output While tlspuffin is fuzzing, lines are printed to the standard output regularly. It is structured according to the following format:

$\langle date \rangle$ [Stats] (GLOBAL) clients: $\langle w \rangle$, corpus: $\langle c \rangle$, obj: $\langle o \rangle$, execs: $\langle i \rangle$, exec/sec: $\langle e \rangle$

where $\langle w \rangle$ is the amount of currently running worker clients, $\langle c \rangle$ is the current size of the corpus, $\langle o \rangle$ is the amount of crashes according to the defined objectives, $\langle i \rangle$ is the total amount of executed iterations and $\langle e \rangle$ is the current speed of the fuzzer in terms of iterations per second. The line above summarizes the current global fuzzing state. Additionally, to this output there is also output for each individual worker client. It uses the (CLIENT) identifier, instead of (GLOBAL) and attaches the edge coverage edges $\langle c \rangle$, where $\langle c \rangle$ denotes how many edges have been covered by this specific worker so far. There is no global edge coverage indication because edge coverage is a local per worker property.

7 Evaluation and Rediscovery of Attacks

In this section we want to evaluate our fuzzer. Because tlspuffin is the first symbolic-model-guided fuzzer, no perfectly comparable baseline exists. Nonetheless, we will compare our approach to an already mentioned bit-level fuzzer [19].

Environment Setup The experiments conducted in this chapter were executed on a server with an `AMD EPYC 7F52`[1] processor with 16 physical cores. The server has 500 GB of memory. During the execution of fuzzing campaigns the server was shared with other users. Because all the experiments use only 20 of the 64 virtual cores, the fact that other users might have caused parallel workload should not impact the statistics shown in this chapter.

Reproducibility Experiments conducted in this chapter are completely reproducible given sufficient hardware resources. We utilize two main tools to achieve. Firstly,

1. Git sub-modules are used to manage dependencies which have been adjusted during the development of tlspuffin.
2. The dependency manager of Rust is used to manage of-the-shelf dependencies, like JSON parsers or serialization libraries.

The most notable dependencies of tlspuffin are the LUTs. The exact LibreSSL and OpenSSL versions are pinned to specific commits within their respective projects. For adjustments in the source code of the LUT, we forked OpenSSL 1.1.1k[2] and LibreSSL 3.3.3[3]. Both dependencies are linked through git sub-modules in the source code repository of tlspuffin. For OpenSSL 1.0.1f and 1.0.2u we reference the upstream repository of OpenSSL[4] directly.

Users of the fuzzer can use the Rust feature flags `openssl101f`, `openssl102u`, `openssl111k` or `libressl` during compilation to create a build which fuzzes the specified LUT. For example, it is very easy to run tlspuffin on OpenSSL 1.0.2u:

```
cargo run --no-default-features --features openssl102u,sancov_libafl,introspection
```

7.1 Evaluation of Fuzzing Campaigns

We conducted several fuzzing campaigns to evaluate tlspuffin. Before we go over vulnerabilities which have been rediscovered through tlspuffin in the next section, we firstly want to provide a few figures which highlight the power of the symbolic-model-guided approach. The data presented here is created by the statistics module of tlspuffin like described in Section 6.8. The following experiments use the default feature options of tlspuffin, unless specified otherwise. If a different LUT is used, then the feature options are adjusted like shown at the beginning of this chapter. The initial seed corpus

[1]`https://www.amd.com/de/products/cpu/amd-epyc-7f52`
[2]`https://github.com/tlspuffin/openssl/tree/fuzz-OpenSSL_1_1_1k`
[3]`https://github.com/tlspuffin/libressl-src/tree/master/libressl`
[4]`https://github.com/openssl/openssl`

is adjusted, such that only supported traces are included. This means for example, that traces which use TLS 1.3 features are excluded, if OpenSSL 1.0.x is used as LUT.

Seed Corpus Size The seed corpus size indicates how many distinct traces have been discovered. The code coverage from Section 6.7.1 is used as metric to determine when two traces are equal. It is

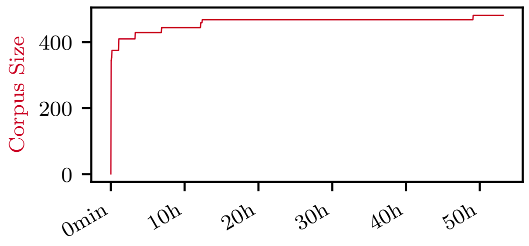


Figure 7.1: Seed Corpus Size while fuzzing OpenSSL 1.1.1k.

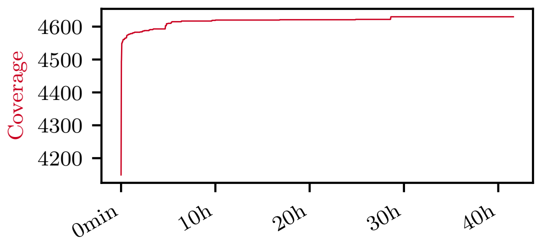
clearly visible in Figure 7.1 that the fuzzer is able to discover many new traces in the very beginning. After the first hour, the rate of newly discovered traces decreases. We can clearly see, that the fuzzer stops showing progress for over 30 hours until it then finds more traces which can be added to the seed corpus. Even though it would be interesting to know why tlspuffin suddenly finds further edges after such a long time, this data is not available, because the seeds in the corpus are currently not persisted together with a timestamp of discovery.

Coverage Code coverage is a valuable criterion which can be used to compare tlspuffin to other fuzzers. Chen, lan, and Venkataramani fuzzed TLS messages of OpenSSL 1.0.x and OpenSSL 1.1.x using AFL [19, 77]. The exact version is not mentioned in the paper. Even though we expect the changes between patch versions like 1.0.1 and 1.0.2 to be minor, they might lead to differences in code coverage because of differences in the total count of edges in the source code. For example, the reason for the difference in the coverage between OpenSSL 1.0.1f and 1.0.2u lie in the difference of total code edges like in can be seen in Figure 7.2.

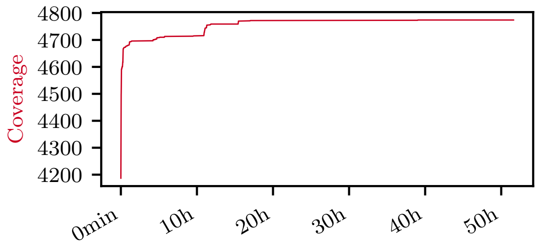
Furthermore, the metric for code coverage depends on the definition on what a code edge is. The authors also used LLVM for instrumentation which means that the definition should be the same as the one of tlspuffin. Chen, lan, and Venkataramani utilized the ASAN instrumentation which we also do in some fuzzing campaigns. It is important to note that the usage of this instrumentation automatically increases the sum of all edges. Therefore, the absolute value of discovered edges is not comparable. Instead, we have to use the ratio between discovered edges and all edges, which offers a slightly better value for comparison. Nonetheless, this value is also not perfectly suited, because it is unclear in which way the instrumentation in [19] changes the edge count.

The reached coverage like depicted in Figure 7.2 is significantly higher than fuzzing a single TLS messages like in [19]. For single messages only a coverage of around 10% is reached. The combined coverage of fuzzing TLS messages in several stages of the TLS handshake as done in [19], comes closer to the coverage of tlspuffin. A coverage of 15.70% for OpenSSL 1.0.x was reached. This Again it is not very clear whether this value is comparable to the coverage reached by tlspuffin, because the authors did not disclose the exact version of OpenSSL they fuzzed.

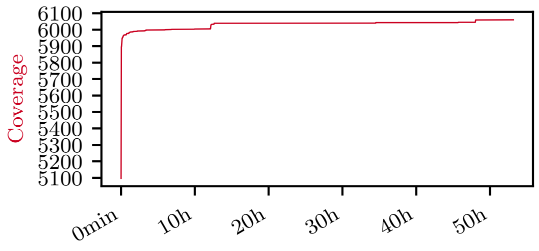
Another factor which should be considered is that it is unclear whether the sets of reached edges from [19] and tlspuffin intersect or are disjoint. We estimate that bit-level fuzzing will discover edges mostly in the implementation of cryptographic primitives. Therefore, a takeaway is that coverages



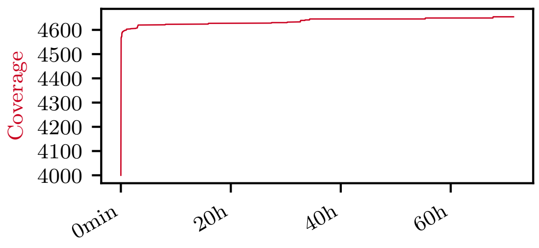
(a) OpenSSL 1.0.1f with 20.47% coverage.



(b) OpenSSL 1.0.2 with 17.26% coverage.



(c) OpenSSL 1.1.1k with 17.29% coverage.



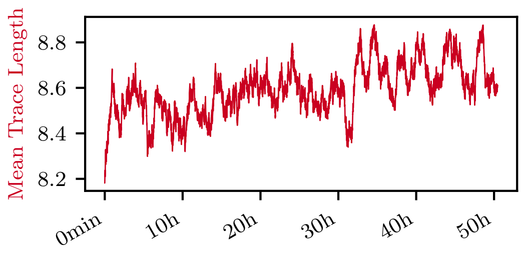
(d) LibreSSL 3.3.3 with 17.72% coverage.

Figure 7.2: Coverage during fuzzing campaigns. The coverage for different LUTs is shown in each subfigure. The discovery behavior of the different LUTs is comparable. In the beginning many new traces are discovered quickly. After the initial phase, the fuzzer discovery new traces slowly. We expect that the uncovered implements features which are not closely related to the TLS logic, like for example the command-line interface.

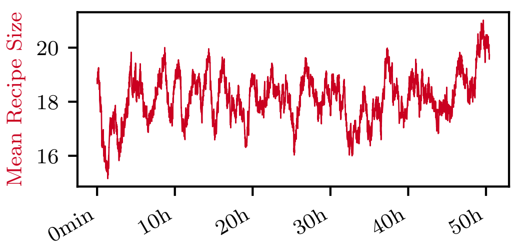
statistics are very difficult to compare. Especially, if the experiments are not reproducible. All experiments of tlspuffin shown in this chapter are reducible like discussed previously.

Mean Trace Length and Recipe Size We can gain additional insights on the shape of traces and terms which are used in fuzzing by observing their length and size. During mutations the trace length and term size are bound. Traces are allowed to shrink and grow between 5 and 15 steps. Terms between 0 and 300 symbols. Figure 7.3 plots the average trace length and term size during a fuzzing campaign for the LUT OpenSSL 1.1.1k. This data gives in sights about which trace lengths and term sizes are used on average while executing traces. It is noteworthy, that there are recipes and traces which use the just introduced limits. For example there are traces of length 15, as well as recipes of size 300. The smoothing of the plot hides this fact.

Executions per Second In order to evaluate the performance of tlspuffin we observed the executions per second during development. By checking this value regularly for significant changes, we could make sure that the performance did not suffer regressions. In Figure 7.4 we can see a major difference in performance when using the ASAN instrumentation or not. When using ASAN, the performance is reduced by a factor of almost 7. This is higher than the estimate of the authors [65]. One reason for this could be that the implementation of tlspuffin is not yet optimized for a reduced amount of heap allocations. Especially, heap allocations are more expensive when ASAN is active.

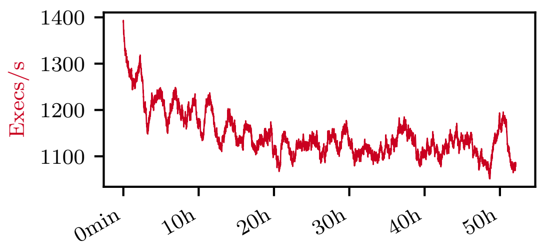


(a) Average trace length

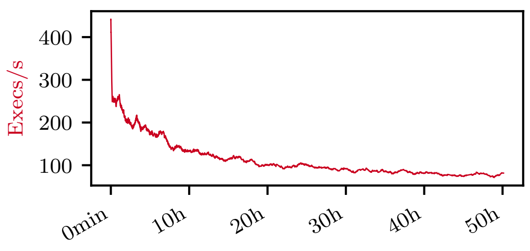


(b) Average recipe size

Figure 7.3: Average trace length and term size during a fuzzing campaign for the LUT OpenSSL 1.1.1k. The average recipe size is calculated across all recipes within every step of every trace. A convolution is used to smooth the values.



(a) OpenSSL 1.1.1k without Sanitizer
(on average 525 executions per second)



(b) OpenSSL 1.1.1k with Sanitizer
(on average 78 executions per second)

Figure 7.4: Executions per second over the duration of a fuzzing campaign. The data shown in this figure has been smoothed using a convolution. The plot on the right shows lower variance, compared to the one on the left. There is no clear explanation to this phenomenon. One reason could be that the introduced memory-management overhead by ASAN dominates other factors in execution time and therefore smooths the curve.

CPU Cycles The profiler introduced in Section 6.8 also profiles in which parts of the fuzzing loop the CPU spends the most time. This also indicates in which areas of the fuzzer major performance improvements can be achieved. The tlspuffin fuzzer spends 94.5% of time executing a trace in OpenSSL 1.1.1. This means that performance optimizations like caching the OpenSSL context created by `SSL_CTX_new` could yield major improvements. Optimizing the mutations or scheduling algorithm in terms of execution performance has lower priority.

7.2 Rediscovery of Attacks

In this section we will go over three known vulnerabilities which have been vetted using tlspuffin. The conducted fuzzing campaigns did not lead to the discovery of new vulnerabilities, but we were able to rediscover already known ones. We actually discovered a bug in the Rust bindings of OpenSSL, which is presented at the very end of this section.

CVE-2021-3449 The CVE-2021-3449[5] represents a denial-of-service attack present in OpenSSL 1.1.1j. Therefore, we fuzzed OpenSSL 1.1.1j instead of 1.1.1k to rediscover this vulnerability. Note that the name of the CVE includes the year 2021, which means that it was discovered just shortly before the start of the development of tlspuffin. This highlights that these kinds of vulnerabilities still exist.

A TLS server using this version of OpenSSL experiences a segmentation fault because of a `NULL` pointer dereference[6]. This attack is possible to detect through tlspuffin, because the fuzzer only has to adapt the client attacker trace from Section 5.3.2 in two steps.

1. Session renegotiation must be enabled by adding an empty Renegotiation Info extension to the initial Client Hello.
2. The initial Client Hello must be duplicated and the content of the Renegotiation Info extension must be set to the transcript hash of the Finished message. The modified Client Hello message must now be encrypted and sent to the server.

The two steps can be represented through a series of **Repeat**, **Replace-Reuse** and **Replace-Match** mutations from Section 5.3.3 to the client attacker seed trace from Section 5.3.2. In order to speed up the discovery of this vulnerability, we already enabled session renegotiation in the client attacker trace for TLS 1.2. The vulnerability was rediscovered after about 9 hours.

The tlspuffin fuzzer was able to automatically find this vulnerability without any major adaption of the fuzzer. Independently of this discovery, similar distinct vulnerabilities can be found. The available mutations or function symbols are not tailored to this vulnerability. The same is true for the next vulnerability which is called FREAK.

FREAK The fuzzer is able to trigger the FREAK vulnerability by creating a MITM scenario in which the attacker downgrades the cipher suite and therefore causes an ephemeral key exchange with 512-bit temporary RSA keys [7]. Unfortunately, we were not able to detect this circumstance as a security violation through our oracle (see Section 5.2). In Section 5.2 we introduced a check for downgrades. With the current definition, we can not detect FREAK, because handshake would need to succeed, before we compare the used cipher suites. If the found attack trace would succeed, then we actually could detect the downgraded cipher suite. In order to finish the trace, the attacker would need to factorize the ephemeral key used in the key exchange. Currently, an attacker in tlspuffin does not have this capability.

Therefore, we need to detect a security violation before the handshake is finished. This is currently not possible with tlspuffin. The reason for this is that an ephemeral RSA key exchange in TLS 1.2 is not a security violation on its own. In fact the server is allows to initiate an ephemeral key exchange after it authenticated against the client. The used key length for ephemeral keys is also not restricted by the specification. The TLS 1.2 specification however determines that ephemeral key exchanges are only allowed if the cipher suite requested one[7] [61, Chapter 7.4.3]. In order to detect this, a security policy which formalizes the just mentioned rule from Chapter 7.4.3 of the TLS 1.2 specification would be required in tlspuffin.

Even though the security violation oracle is not able to detect it automatically, we still rediscovered it by manually tweaking the source code of OpenSSL. Because we know that the FREAK vulnerability is present in OpenSSL 1.0.1f, we were able to manually implement a crash when an ephemeral key exchange happens, even though no EXPORT cipher is used. That way we were able to test whether mutations can lead to a scenario in which FREAK can be exploited.

[5]`https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3449`
[6]`https://github.com/openssl/openssl/commit/fb9fa6`
[7]In fact the TLS 1.2 also allows ephemeral key exchanges for cipher suites which specify anonymous key exchanges.

FREAK can be fixed by disallowing ephemeral key exchanges when a cipher suite is used which uses RSA as key exchange method. This includes the EXPORT cipher suites which for limit the RSA key length to 512 bits. FREAK has been fixed in OpenSSL by allowing ephemeral RSA key exchanges only for insecure EXPORT cipher suites[8].

Heartbleed In order to trigger the well known Heartbleed vulnerability [31], we first had to implement Heartbeat messages, because they are not available in the TLS library rustls. The Heartbleed vulnerability depends on an inconstancy between the advertised length of an array and the actual length of the array. A TLS Heartbeat request and response message contains a single array. This array is encoded by encoding the length as unsigned 16-bit integer using the big-endian format and then appending each byte of the array. An OpenSSL 1.0.f TLS server performs an out-of-bound read in the array which was sent by the client and sends back all read data. The response could contain any data which happens to be laid out near the initially received array in memory. This memory over-read is detectable because of the memory sanitization using ASAN (see Section 6.7.3).

In order to model this inconsistency tlspuffin was given the capability to use a fake length in Heartbeat messages. This is contrary to the design idea of tlspuffin, because we did not intend to fuzz on a bit-level, but instead fuzz messages and their fields. Nonetheless, this capability was added such that we could verify tlspuffin and its mutations in the early development of the fuzzer.

In fact, tlspuffin was able to discover Heartbleed within the first minute. This is very impressive if compared to AFLGo which takes about 20 minutes to find Heartbleed [16]. One could argue that, that tlspuffin experiences over-fitting because we added the capability to use a fake length for arrays in the Heartbeat message. The capability could be seen as being general-purpose through. Buffer over-reads or overflows are common in non-memory safe languages.

It is likely that tlspuffin is able to find more variants of Heartbleed. The reason for this is that encrypted Heartbeat messages can be sent later in the handshake. In contrast to classical bit-level fuzzing, fuzzing encrypted messages is in the scope of tlspuffin.

Memory Leak in Rust FFI Even though this is not a security related attack, we discovered a bug in the Rust Foreign Function Interface (FFI) of OpenSSL 1.0.x. Unlike the previous vulnerabilities this bug is actually a new discovery instead of a rediscovery. The Rust API for enabling ephemeral RSA key exchanges forces memory leaks within OpenSSL. Because the `unsafe` keyword is used for FFIs, the memory safety guarantees are disabled for interface between Rust and C. The issue was reported to the developer of the FFI[9]. Because the RSA ephemeral key exchanges are only used for insecure EXPORT cipher suites the resolution of this bug is to remove the API.

[8]`https://github.com/openssl/openssl/commit/ce325c6`
[9]`https://github.com/sfackler/rust-openssl/issues/1529`

8 Conclusions & Future Work

In the following section we will present our findings and draw conclusions from them. After that, we will present the key points in which tlspuffin could benefit the most from improvements.

8.1 Findings

In this thesis, we apply a novel technique to reach deep states of cryptographic protocols. By guiding the fuzzing process through a symbolic model, we are able to reach high coverage quickly. The main contributions of this thesis are:

- Adoption of a formal model for protocol verification for fuzzing;
- Design of a fuzzer capable of using this formal model;
- Reference implementation for the design in order to prove its feasibility;
- Evaluation by rediscovering known vulnerabilities and discovering a novel memory leak.

To our knowledge, this is the first work on protocol fuzzing utilizing a symbolic Dolev-Yao model [30] to define the input space of our fuzzer. The initial goal of fuzzing implementations of the formally verified TLS protocol by guiding the process through a symbolic model is fulfilled. On this basis, we conclude that by drawing inspiration from a well-known formal model and then creating a fuzzing-oriented variant of it, significantly contributed to the mature design of the novel fuzzer. In fact, this methodology of using formal methods corresponds to the best practices used in requirements engineering.

Design Phase In the design phase of tlspuffin we explored which components are required to develop a fully-automated fuzzer. We introduced an evaluation based fuzzing algorithm which uses unique and domain-specific mutations and initial seeds. A novel security related instrumentation bug oracle is presented, which allows the discovery of security related attacks like security downgrades, or authentication bypasses. We also shed light on various challenges like the robustness of traces or the explosion of recipe sizes. Concrete solutions to these problems have been introduced. We also showed that it is possible to design a bug oracle which does not only check for crashes, but also checks security properties like authentication or downgrade resistance. A promising aspect of this design phase is, that not only traces according to the TLS specification can be constructed, but also deviant attack traces.

Implementation Phase Based on the design for a symbolic-model guided fuzzer, we developed a concrete reference implementation. Solutions to challenges which were identified in the design of tlspuffin have been implemented. Various implementation challenges have been identified. For example like the non-trivial data structure of terms, the not to underestimate effort of providing manual and concrete implementations for function symbols, or the implementation of checks for security properties. The findings from the implementation phase include that the development of a symbolic-model-guided fuzzer is feasible. We also contributed to the LibAFL framework [33] in a non-trivial manner. We contributed 5 patches and raised 8 issues.

Evaluation Phase To better understand the behavior of the fuzzer during a fuzzing campaign, evaluation tools have been developed. Similar to the programs which are being tested through fuzzing, the fuzzer itself can also seem like a black-box. Therefore, we contributed a toolset to profile and analyze the inner workings of the fuzzer during experiments. Despite many other fuzzers miss such a toolset, we conclude that it is a helpful instrument during development and evaluation of a fuzzer.

Moreover, further experiments showed, that the fuzzer is in fact able to discover vulnerabilities on its own. Despite the fact that tlspuffin only rediscovered known vulnerabilities, there is proof that the approach is capable of finding vulnerabilities. It is especially noteworthy that the fuzzer discovered the Heartbleed vulnerability in a fraction of the time which classical bit-level fuzzer require. Importantly, the experiments also showed that vulnerabilities like CVE-2021-3449 which are only discoverable by sending encrypted messages are in scope. Generating encrypted message is out-of-scope for usual bit-level fuzzers. The fuzzer is also able to mutate a trace such that it creates a scenario in which the FREAK vulnerability is triggered. The automatic detection of FREAK was not possible with the current implementation of the security violation oracle. A better implementation was out-of-scope in this thesis because of time constraints. It is noteworthy, that aside from the goal to find security related vulnerabilities, some fuzzing campaigns lead to the discovery of a bug which causes memory leaks in the Rust bindings for OpenSSL.

The present findings from the experiments suggest that the approach has advantages over classical bit-level fuzzing like the possibility to fuzz encrypted messages. The fact that no new vulnerabilities have been discovered suggests that either no logical security vulnerabilities are present in TLS implementations or that the capabilities of the implemented symbolic model are too limited. In the following section we will discuss the key points in which more work is required to improve on the current design and implementation of a symbolic-model-guided fuzzer.

8.2 Future Work

Because symbolic-model-guided fuzzing is a novel idea, there are many areas which could benefit from improvements. In the following we will discuss major opportunities for future research and work in the implementation and design of tlspuffin.

8.2.1 Implementation Improvements

Due to time constraints while developing tlspuffin and writing this thesis several improvements of the fuzzer implementation could not be conducted. The following paragraphs will give insights about open points.

Improved Profiling & Evaluation The evaluation of tlspuffin raised a major question: Why is the fuzzer only showing marginally small progress after the initial discovery phase? The answer to this question requires more data and statistics about the created offspring while mutating traces. Right now we only have data bout the minimum, maximum, and mean trace length and recipe size like shown in Section 7.1. Therefore, future work could focus on improving the extent of profiling within tlspuffin. Possible further statistics could include:

- the usage of function symbols, which yield a successfully executing trace,
- the usage of mutations operations, or
- the success rate of applying recipe mutations to terms.

All of these statistics require profiling the trace execution. Furthermore, specific profiling frameworks could be used to ease the implementation.

Improved Fuzzer Feedback Throughout the thesis, we only use code edge coverage as a metric for feedback in the genetic fuzzing algorithm. This means the only score for how fit a trace is, is determined by its code coverage. Other metrics have already been proposed like in [16] which utilizes a metric which determines how far the fuzzer is from reaching certain points in the source code of the LUT. Future research is required to find a metric which matches the nature of protocols. For example a metric could score traces based on their convergence or divergence to the TLS specification. Moreover, traces could be rated, based on new TLS extensions they discovered through fuzzing.

Furthermore, feedback like whether the trace resulted in a successful handshake or not should be considered. Moreover, errors by the LUT can be used to gain additional feedback [56]. Unexpected behavior by the LUT like the sending of malformed messages could be useful.

Implement further Mutations In Section 5.3.3 we introduced a lot of mutations but only implemented a small subset of them due to time constraints. Further evaluation of the usefulness of implemented and not yet implemented mutations, could give insight about further helpful mutations. For example the fuzzer could be profiled in a way which tracks which series of mutations are able to yield interesting traces. The implementation of further mutations could provide a better code edge coverage or even yield new vulnerability discoveries.

Implement further Function Symbols Each and every function symbol introduced in tlspuffin requires a concrete implementation. Because of time constraints, not all reasonable function symbols have been added. Further work could focus on implementing new ones, or changing present ones in such a way that more meaningful traces can be represented. The current implementations are mostly based on lists of TLS messages and extensions. By creating more seed traces which cover more features of the TLS specification, ideas for further function symbol can be acquired. An example for this would be a trace in which the TLS server is the attacker. This would complement the client attacker trace introduced in Section 5.3.2.

Explore LibAFL Framework The used framework for implementing the fuzzer is still in an early stage of development. This means that we had to overcome many challenges in order to utilize LibAFL. The benefits of using a framework to implement the fuzzing algorithm have not been used to a full extend. For example, it is possible to benefit from different schedulers, feedbacks, or objectives. So far only a very default configuration of the framework has been tested. Because LibAFL is actively developed and new implementations for building blocks are being released on a regular basis, the development of tlspuffin could benefit from them.

8.2.2 Design Improvements

The design of tlspuffin could be improved in two distinct ways. Both open points already have been mentioned during the thesis. We will revise them here to make clear in which directions further work is required.

Trace Robustness In Section 5.4.1 we discovered the problem of robustness. Mutations of traces can lead to invalid traces. Therefore, we introduced robust queries which match acquired knowledge. Unfortunately, the type of messages is sometimes only obvious from its context. For example in TLS 1.3 the majority of messages sent during a handshake are encrypted messages. All encrypted message have the same type. Therefore, queries could be extended by contextual knowledge. This means that messages are not only classified based on their obvious binary-encoded type, but also on the context in which they appear.

Improved Security Violation Oracle In Section 5.2 we designed a bug oracle which is able to check security properties. Right now we only check after successful handshakes whether no security violation happened. For example whether a downgrade or authentication bypass happened. This could be extended by also verifying invariants during the execution of a handshake. That way we could for example detect the FREAK vulnerability automatically.

Furthermore, the security violation oracle could also be extended to other TLS implementations. Right now only OpenSSL 1.1.1k is supported. Support for checking security violations could be ported to LibreSSL 3.3.3 and backported to OpenSSL 1.0.x.

8.2.3 New Directions

Finally, we could also extend tlspuffin in different and novel directions. The following we will provide two new ideas which could lead to new discoveries.

Protocol Adoption In order to proof the adaptability of symbolic-model-guided fuzzing, the approach could be adapted to a different protocol like QUIC [41] which has similar goals like TLS. This represents a challenge because of the high development effort of function symbol implementation. Furthermore, the LUT harness would need to be adjusted for a different protocol. For example, the knowledge queries would need to be modified, because they contain a domain-specific TLS message type right now. Nonetheless, the general design of tlspuffin already is generic such that it can be applied to arbitrary cryptographic protocols. For example the notion of terms, traces, and their semantics should be reusable. Moreover, the fuzzing algorithm is also applicable to different protocols.

Cross-implementation differential Fuzzing Lastly, we mentioned in Table 6.1 that we are able to run traces in an environment in which different LUTs are active at the same time. For example, it could be interesting to run a trace against an OpenSSL and LibreSSL server. That way, we could detect differences in their behavior and gain additional feedback. For example traces which yield a different behavior for different LUTs could be rated as being interesting. Differential fuzzing could lead to the discovery of further logical bugs in implementations.

Bibliography

[1] C. Allen and T. Dierks. *The TLS Protocol Version 1.0.* RFC 2246. Jan. 1999.

[2] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, et al. "The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications." In: *Proceedings of the 2008 International Conference on Computer-Aided Verification.* Springer, 2005, pp. 281–285.

[3] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. "Ijon: Exploring Deep State Spaces via Fuzzing." In: *Proceedings of the 2020 Symposium on Security and Privacy.* IEEE, 2020, pp. 1597–1612.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That.* 1st ed. Cambridge University Press, 1998.

[5] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. "SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr." en. In: *Proceedings of the 2006 International Conference of Information Security.* Lecture Notes in Computer Science. Springer, 2006, pp. 343–358.

[6] D. Benjamin, B. Smith, and A. Langley. *ring: Safe, fast, small crypto using Rust.* 2021. URL: `https://github.com/briansmith/ring` (visited on 09/15/2021).

[7] 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: *Proceedings of the 2015 Symposium on Security and Privacy.* IEEE, 2015, pp. 535–552.

[8] B. Beurdouche, A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and K. Bhargavan. "FLEXTLS: A Tool for Testing TLS Implementations." In: *Proceedings of the 2015 Workshop on Offensive Technologies.* USENIX, 2015.

[9] M. Bezem, J. W. Klop, and R. d. Vrijer, eds. *Term rewriting systems.* Cambridge Tracts in Theoretical Computer Science 55. Cambridge University Press, 2003.

[10] K. Bhargavan, B. Blanchet, and N. Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate." In: *Proceedings of the 2017 Symposium on Security and Privacy.* IEEE, 2017, pp. 483–502.

[11] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. "Cryptographically verified implementations for TLS." In: *Proceedings of the 2008 Conference on Computer and Communications Security.* ACM, 2008.

[12] J. Birr-Pixton, D. Ochtman, and B. Smith. *rustls: A modern TLS library in Rust.* 2021. URL: `https://github.com/rustls/rustls` (visited on 09/15/2021).

[13] B. Blanchet. "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif." In: *Foundations and Trends in Privacy and Security* 1.1-2 (2016), pp. 1–135.

[14] B. Blanchet. "Security Protocol Verification: Symbolic and Computational Models." en. In: *Proceedings of the 2012 International Conference on Principles of Security and Trust.* Lecture Notes in Computer Science. Springer, 2012, pp. 3–29.

[15] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre. *ProVerif 2.02pl1: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial.* 2020.

[16] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. “Directed Greybox Fuzzing.” In: *Proceedings of the 2017 Conference on Computer and Communications Security.* ACM, 2017.

[17] M. Böhme, V.-T. Pham, and A. Roychoudhury. “Coverage-based Greybox Fuzzing as Markov Chain.” In: *Proceedings of the 2016 Conference on Computer and Communications Security.* ACM, 2016.

[18] R. Bressler. *Proposed Moratorium on Changes to Network Protocol.* RFC 72. Sept. 1970.

[19] Y. Chen, T. lan, and G. Venkataramani. “Exploring Effective Fuzzing Strategies to Analyze Communication Protocols.” In: *Proceedings of the 2019 Workshop on Forming an Ecosystem Around Software Transformation.* ACM, 2019.

[20] V. Cheval, S. Kremer, and I. Rakotonirina. “DEEPSEC: Deciding Equivalence Properties in Security Protocols Theory and Practice.” In: *Proceedings of the 2018 Symposium on Security and Privacy.* IEEE, 2018, pp. 529–546.

[21] V. Cortier, S. Delaune, and P. Lafourcade. “A survey of algebraic properties used in cryptographic protocols.” In: *Journal of Computer Security* 14 (2006), pp. 1–43.

[22] V. Cortier and S. Kremer. “Formal Models and Techniques for Analyzing Security Protocols: A Tutorial.” en. In: *Foundations and Trends in Programming Languages* 1.3 (2014), pp. 151–167.

[23] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. “A Comprehensive Symbolic Analysis of TLS 1.3.” In: *Proceedings of the 2017 Conference on Computer and Communications Security.* ACM, 2017.

[24] C. J. F. Cremers. “The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols.” In: *Proceedings of the 2008 International Conference on Computer-Aided Verification.* Springer, 2008, pp. 414–418.

[25] *CVE-2014-0160.* Available from MITRE, CVE-ID CVE-2014-0160. Mar. 2021. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3449` (visited on 06/03/2021).

[26] J. De Ruiter and E. Poll. “Protocol state fuzzing of TLS implementations.” In: *Proceedings of the 2015 Security Symposium.* SEC’15. USENIX, 2015, pp. 193–206.

[27] W. Diffie and M. Hellman. “New directions in cryptography.” In: *Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[28] E. W. Dijkstra. “On the Role of Scientific Thought.” In: *Selected Writings on Computing: A personal Perspective.* Springer, 1982, pp. 60–66.

[29] S. Dinesh, N. Burow, D. Xu, and M. Payer. “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization.” In: *Proceedings of the 2020 Symposium on Security and Privacy.* IEEE, 2020.

[30] D. Dolev and A. Yao. “On the security of public key protocols.” In: *Transactions on Information Theory* 29.2 (1983), pp. 198–208.

[31] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, et al. “The Matter of Heartbleed.” In: *Proceedings of the 2014 Conference on Internet Measurement Conference.* ACM, 2014.

[32] P. Eronen, H. Tschofenig, H. Zhou, and J. A. Salowey. *Transport Layer Security (TLS) Session Resumption without Server-Side State.* RFC 5077. Jan. 2008.

[33] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. “AFL++ : Combining Incremental Steps of Fuzzing Research.” In: *Proceedings of the 2020 Workshop on Offensive Technologies.* USENIX, 2020.

[34] E. R. Gansner and S. C. North. “An Open Graph Visualization System and Its Applications to Software Engineering.” In: *Software Practice Experience* 30.11 (Sept. 2000), pp. 1203–1233.

[35] P. Godefroid. “Random testing for security.” In: *Proceedings of the 2007 International Workshop on Random testing.* ACM, 2007.

[36] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, et al. “Array programming with NumPy.” In: *Nature* 585.7825 (2020), pp. 357–362.

[37] K. Headley. “A DSL embedded in Rust.” In: *Proceedings of the 2018 Symposium on Implementation and Application of Functional Languages.* ACM, 2018.

[38] B. Heisler. *criterion.rs: Advanced Fuzzing Library.* 2021. URL: `https://github.com/bheisler/criterion.rs` (visited on 09/15/2021).

[39] J. D. Hunter. “Matplotlib: A 2D graphics environment.” In: *Computing in Science and Engineering* 9.3 (2007), pp. 90–95.

[40] *Information technology - Software failures and their assessment by suppliers and customers.* DIN. 1995.

[41] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport.* RFC 9000. May 2021.

[42] H. Kario. *tlsfuzzer: SSL and TLS protocol test suite and fuzzer.* 2015. URL: `https://github.com/tlsfuzzer/tlsfuzzer` (visited on 09/15/2021).

[43] D. H. Krawczyk and P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF).* RFC 5869. May 2010.

[44] A. Lee, M. Smid, and S. Snouffer. *Security Requirements for Cryptographic Modules.* en. May 2001.

[45] G. Lowe. “Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR.” In: *Proceedings of the 1996 International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 1996, pp. 147–166.

[46] D. Maier and A. Fioraldi. *LibAFL: Statistics-driven benchmarking library for Rust.* 2021. URL: `https://github.com/AFLplusplus/LibAFL` (visited on 09/15/2021).

[47] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. “Fuzzing: Art, Science, and Engineering.” In: *CoRR* abs/1812.00140 (2018). arXiv: `1812.00140`.

[48] N. D. Matsakis and F. S. K. II. “The rust language.” In: *Proceedings of the 2014 Annual Conference on High Integrity Language Technology.* ACM, 2014, pp. 103–104.

[49] S. Meier, B. Schmidt, C. Cremers, and D. Basin. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols.” In: *Proceedings of the 2008 International Conference on Computer-Aided Verification.* Springer, 2013, pp. 696–701.

[50] B. P. Miller, L. Fredriksen, and B. So. “An empirical study of the reliability of UNIX utilities.” In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[51] B. P. Miller, M. Zhang, and E. R. Heymann. “The Relevance of Classic Fuzz Testing: Have We Solved This One?” In: *CoRR* abs/2008.06537 (2020). arXiv: `2008.06537`.

[52] J. Munns. *postcard: A serde compatible message library for Rust.* 2021. URL: `https://github.com/jamesmunns/postcard` (visited on 09/15/2021).

[53] G. J. Myers, T. F. Badgett, and C. Sandler. *The Art of Software Testing.* Wiley, 2012.

[54] R. M. Needham and M. D. Schroeder. “Using encryption for authentication in large networks of computers.” In: *Communications of the ACM* 21.12 (1978), pp. 993–999.

[55] M. Nemec, D. Klinec, P. Svenda, P. Sekan, and V. Matyas. “Measuring Popularity of Cryptographic Libraries in Internet-Wide Scans.” In: *Proceedings of the 2017 Annual Computer Security Applications Conference.* ACM, 2017.

[56] V.-T. Pham, M. Böhm, and A. Roychoudhury. “AFLNET: A Greybox Fuzzer for Network Protocols.” In: *Proceedings of the 2020 International Conference on Software Testing, Validation and Verification.* IEEE, 2020, pp. 460–465.

[57] D. A. Plaisted. “Equational Reasoning and Term Rewriting Systems.” In: *Handbook of Logic in Artificial Intelligence and Logic Programming (Vol. 1).* Oxford University Press, 1993, pp. 274–364.

[58] E. Poll, J. de Ruiter, and A. Schubert. “Protocol State Machines and Session Languages: Specification, implementation, and Security Flaws.” In: *Proceedings of the 2015 Security and Privacy Workshops.* IEEE, 2015.

[59] J. Postel. *DoD standard Transmission Control Protocol.* RFC 761. Jan. 1980.

[60] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3.* RFC 8446. Aug. 2018.

[61] E. Rescorla and T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2.* RFC 5246. Aug. 2008.

[62] J. K. Reynolds and D. J. Postel. *Instructions to RFC Authors.* RFC 2223. Oct. 1997.

[63] G. Roy. *Symbolic-Model-Aware Fuzzing.* 2020.

[64] S. Santiago, S. Escobar, C. Meadows, and J. Meseguer. “A Formal Definition of Protocol Indistinguishability and Its Verification Using Maude-NPA.” In: *Proceedings of the 2014 International Workshop on Security and Trust Management.* Springer, 2014, pp. 162–177.

[65] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. “AddressSanitizer: A Fast Address Sanity Checker.” In: *Proceedings of the 2012 Annual Technical Conference.* USENIX, 2012, pp. 309–318.

[66] Y. Sheffer, R. Holz, and P. Saint-Andre. *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS).* RFC 7457. Feb. 2015.

[67] J. Somorovsky. “Systematic Fuzzing and Testing of TLS Libraries.” In: *Proceedings of the 2016 Conference on Computer and Communications Security.* CCS ’16. ACM, 2016, pp. 1492–1504.

[68] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” In: *Proceedings of the 2016 Network and Distributed System Security Symposium.* The Internet Society, 2016.

[69] D. Tolnay and E. Tryzelaar. *Serde: Serialization framework for Rust.* 2021. URL: `https://github.com/serde-rs/serde` (visited on 09/15/2021).

[70] P. Tsankov, M. T. Dashti, and D. Basin. “SECFUZZ: Fuzz-testing security protocols.” In: *Proceedings of the 2012 International Workshop on Automation of Software Test.* IEEE, 2012.

[71] J. S. Vitter. “Random sampling with a reservoir.” In: *Transactions on Mathematical Software* 11.1 (1985), pp. 37–57.

[72] G. Vranken. *cryptofuzz: Differential cryptography fuzzing.* 2019. URL: `https://github.com/guidovranken/cryptofuzz` (visited on 09/15/2021).

[73] O. Way, M. Ray, S. Dispensa, and E. Rescorla. *Transport Layer Security (TLS) Renegotiation Indication Extension*. RFC 5746. Feb. 2010.

[74] N. Williams. *JavaScript Object Notation (JSON) Text Sequences*. RFC 7464. Feb. 2015.

[75] W. Xu, S. Kashyap, C. Min, and T. Kim. "Designing New Operating Primitives to Improve Fuzzing Performance." In: *Proceedings of the 2017 Conference on Computer and Communications Security*. ACM, 2017.

[76] J. Yanovski, H.-H. Dang, R. Jung, and D. Dreyer. "GhostCell: separating permissions from data in Rust." In: vol. 5. ICFP. ACM, 2021, pp. 1–30.

[77] M. Zalewski. *American Fuzzy Lop - Whitepaper*. 2016. URL: `https://lcamtuf.coredump.cx/afl/technical_details.txt` (visited on 09/16/2021).