Radboud University Nijmegen

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Extending Puffin and Dolev-Yao Based Fuzzing to MbedTLS

Master Thesis Computing Science

Author:
Toon Lenaerts

Supervisor:
prof. dr. Frits Vaandrager

Second Reader:
dr. ir. Erik Poll

March 2026

Abstract

To improve confidence in a secure internet, we applied a novel fuzzing tool, Puffin, to MbedTLS, a commonly used implementation of the TLS protocol. Puffin uses the Dolev-Yao framework to extend its capabilities: it can discover not only memory management faults, but faults in the implementation of protocol logic as well.

In this work we have tested the capabilities of the Puffin fuzzer to find these different types of faults in MbedTLS. For this, we have written our own harness which communicates packets that Puffin generates with the MbedTLS library and back. We have implemented Puffins claim extraction framework inside the code of MbedTLS, which allows the Puffin to reason over MbedTLSs internal information. Additionally, we extended the description of Puffins Dolev-Yao model with a framework for information extraction. We have built a suite of insertable defects for MbedTLS which allow us to test what types of defects Puffin is able to find, and have found that Puffin is able to correctly find memory management defects and a subset of implementation-level protocol logic defects. Finally we have executed Puffin on a version of MbedTLS without inserted errors, where no new defects were found.

Contents

1 Introduction

In our digital age, we cannot understate the importance of a secure internet. When the protocols at the foundation of our digital infrastructure are compromised, the impact is massive [13]. Transport Layer Security (TLS) is one such protocol. It provides confidentiality, authenticity and integrity for protocols as HTTPS, OpenVPN, SMTP, FTP and more. As such, much effort has been undertaken to test and verify both the TLS specification [8, 36, 27] and its implementations [2, 5, 35, 22]. Regardless, the protocol and implementing programs are complex, and errors are still found [17].

MbedTLS [1] is a well-known implementation of TLS specifically aimed at microcontrollers, and therefore it sports an efficient memory-usage and small code footprint. The MbedTLS library, however, is not limited to just IoT-devices and microcontrollers, and can be used for desktop applications as well. One such example is OpenVPN-NL, a hardened version of the much used OpenVPN software, commissioned by the Dutch government, wherein OpenSSL was replaced with MbedTLS, as it was deemed more secure. Considering the impact of a possible fault in a government-used VPN program, more research into this choice is necessary.

One technique that is commonly used in modern software testing is fuzzing [2, 3, 4, 15, 16, 20, 22, 28, 35]. Simply put, fuzzing is the process of testing a System Under Test (SUT) by taking a normal input and randomly mutating it. This mutated input is executed, after which it is checked whether the execution produced unwanted behaviour. Any input that causes such behaviour is analysed, which should lead to the fault at the root of the problem. Fuzzers range in complexity, some mutate input by editing known correct input, while others can leverage feedback from the SUT to steer the mutations they perform, or even infer a state machine of the SUT [10]. As such, fuzzing can range in knowledge of and access to the SUT, ranging from a black-box setting to analyzing source code. Regardless of these variations, fuzzing has shown to be most useful at finding a particular type of fault, memory management faults [30].

Ammann et al. [2] created Puffin, a gray-box fuzzer for TLS, which gains extensive knowledge of a protocol by specifying input flows in the Dolev-Yao attacker model (Section 4.3). Through the use of this model, Ammann et al. reports that Puffin is not only capable of efficiently finding memory management faults but also can find implementation-level protocol logic attacks. These attacks stem from faults in the SUT that break the guarantees provided by the protocol. For example, take a program that guarantees confidentiality, but some sequence of input causes it to transmit plaintext instead. This mistake does not have to be related to memory mismanagement, and will be missed if a fuzzer only checks for timeouts and program crashes. Ammann et al. implemented this idea into a fuzzer for three TLS implementations: OpenSSL, WolfSSL and BoringSSL.

In this work, we extend the Puffin fuzzer to test MbedTLS. The code of the project is available on GitLab[2]. We analyse MbedTLS with Puffin, in order to increase trust in whether this library is able to provide the confidentiality, authenticity and integrity properties that the TLS specification gives, and thereby trust in these properties of the applications it is used in. We intend to do this through answering the following research questions:

RQ1. What memory management faults is Puffin able to detect in MbedTLS?

RQ2. What implementation-level protocol logic faults is Puffin able to detect in MbedTLS?

RQ3. Which existing memory management faults and implementation-level protocol logic faults does Puffin find in MbedTLS?

[1] `https://www.trustedfirmware.org/projects/mbed-tls/`
[2] `https://gitlab.science.ru.nl/alenaerts/MbedTLSPuffin`

In this thesis, we present our methods, results and conclusions towards answering these research questions. First, we examine works related to our research in Section 2. Then, we describe our testing domain, TLS and its security properties in Section 3. In Section 4, first, we describe in detail the two kinds of defects we intend to find, how Puffin and the Dolev-Yao model it uses should enable us to find the defects. In Section 5 we describe how we extended Puffin to be able to fuzz MbedTLS. In Section 6 we describe how we verify that Puffin can find the different type of defects in MbedTLS, thus answering our research questions. Questions **RQ1** and **RQ2** we answer through mutation testing: inserting a sample of types faults into MbedTLS and verifying whether Puffin is able to find them. Once we know that Puffin is capable of detecting these categories of faults, we test whether these faults are present in a clean instance of MbedTLS. As there is a chance that these faults are not actually present in MbedTLS, this setup shows us that if such faults were present, Puffin would have been able to find them, thus answering the other question **RQ3**. Then, we show the results of our fuzzing campaigns. In Section 7 we analyse these results and discuss choices we made with impact on these results. Then, we present our conclusions and answers to the research questions in Section 8. Finally, in Section 9 we provide angles for further research in this area of fuzzing.

2 State of the Art

In this section, we discuss existing research that is related to our work. As TLS is a commonly used protocol, a significant number of works have been published on its security. These works focus on different aspects of TLS' security and as such employ varying techniques. In Section 2.1 we expand upon such research. Then, in Section 2.2, we show research into three commonly used techniques to test protocol implementations: fuzz testing (Section 2.2.1), symbolic verification (Section 2.2.2) and model learning in the context of protocol correctness (Section 2.2.3). In these sections, we also note how said techniques have been applied to TLS. We conclude every section in this chapter with a note on how the discussed research relates to our research questions.

2.1 TLS security

As TLS is a widely used protocol, there is much research performed on its security and weaknesses; some famous examples of problems being the Heartbleed bug [13] and the POODLE [21] attack.

The work most related to this thesis is the Puffin fuzzer by Ammann et al. [2], which asks the user to define the happy-flow of a protocol in the term algebra of the Dolev-Yao attacker model. The fuzzer then mutates inputs based on the variables and fields of the given data flow. The program can extract security properties from both the user-provided happy-flow and stored runs on the SUT based on what data has been shared between agents. Which allows the fuzzer to find a different class of vulnerabilities than the fuzzers Puffin is compared to. Ammann et al. use the fuzzer on the TLS protocol and specifically the OpenSSL and wolfSSL implementations. Four new vulnerabilities in wolfSSL were found. In the work, Ammann et al. describe that Puffin can be extended to other TLS implementations or other protocols. As of writing, an extension to SSH is a work-in-progress. However, no other TLS implementation have been tested. Additionally, the work does not clearly explanation the information extraction from data in the Dolev-Yao model.

Surveys on TLS security have been performed by Swierzy et al. [30] and Alnahawi et al. [1]. Swierzy et al. [30] provide an overview of automated methods to test TLS. They categorize existing tools into eight approaches, including fuzzing [2, 35, 22], automata learning [26] and model-based testing, and describe for each of these approaches the types of vulnerabilities they are best suited to finding, and how much knowledge a test developer should have of the system under testing. They note that few of the selected tools focus on just one of the bug-finding approaches, but often a combination. Nevertheless, the tools are deemed relatively small in scope, as they often focus on only parts of a TLS library. Another observation made is a split between high-effort, specifically focused tools used by security researchers and (relatively) low-effort, tools with a wider approach from library developers. Alnahawi et al. [1] provides a literature review over research on post-quantum security of TLS, specifically the confidentiality and authenticity that the protocol should provide. Perhaps the most authoritative source on TLS security requirements is provided by the TLS 1.2 and 1.3 RFCs [24, 25] themselves: Appendices F and E respectively provide plain text overviews.

More examples of fuzzing research on TLS are:

- Walz and Sikora's [35] differential tester uses a tree structure to represent a TLS packet. Mutations are performed to this tree and then partially repaired, altering the resulting packet. The setup then sends these tests to a number of different TLS implementations, aiming to receive several different responses.
- Pan et al. [22] uses the idea behind the differential tester of Walz et al. [35] and

the NEZHA differential testing framework by Petsios et al. [23] to more efficiently find discrepancies in OpenSSL, BoringSSL and LibreSSL.

- Hu et al. [15] applied a tree-based mutation strategy to three stateful fuzzers, in order to improve code and state coverage whilst fuzzing the OpenSSL TLS server.

These works provide an overview of the security properties that TLS should provide (see Section 3.3), and thus they provide information on what properties a implementation-level protocol defect should break. As such, this research is related to our second research question **RQ2**.

2.2 Testing techniques

In this section, we discuss research on three techniques that are often used to test implementations of protocols. The articles discussed are mostly general works on the techniques and some works on applying said techniques on TLS. The methods we discuss are fuzz testing (Section 2.2.1), symbolic proofs (Section 2.2.2) and model learning (Section 2.2.3).

2.2.1 Fuzz testing

Ever since its introduction by Miller et al. [20], fuzzing has been extensively researched: Daniele et al. [10] provides an overview of existing stateful fuzzers for generic protocols; fuzzers which can be adapted to work for different protocols without much issue. The standard classification between black-box, grey-box and white-box fuzzers is foregone for a two-pronged distinction between fuzzers that are grammar-based or grammar-learners, and between evolutionary and non-evolutionary fuzzers. A grammar-based fuzzer requires the user to define a grammar for generated inputs, while a grammar-learner can infer this from various sources. An evolutionary fuzzer uses some form of feedback system to guide its mutating of inputs. An example for some of the given categories:

- AspFuzz from Kitagawa et al. [16], a grammar-based fuzzer, requires a user to specify an application layer protocol in a specific "tsfrule" format, which defines both a message-level language through regular expressions and a protocol-level state machine through transition rules.

- Bastani et al.'s [4] GLADE is a grammar learner fuzzer that uses a set of hand-picked input seeds to construct a context free grammar in two phases: First it creates a regular expression from the seed, testing on the SUT if parts of the seeds contain loops, and then changing the regular expression into a context free grammar. This is done though looking back at the steps used to minimize the input in the first phase, and creating a rule for every (useful) step.

- Nyx-Net by Schumilo et al. [28] is an evolutionary fuzzer that uses snapshots to repeat tests at certain trace depths, which it can create rapidly. By changing when to take these snapshots, various strategies in fuzzing can be inferred.

- RESTler is an evolutionary grammar based fuzzer by Atlidais et al.[3]. It uses the definition of a REST API to generate a grammar which is then used to send mutated messages to a server.

Regarding our research questions, this angle of research is relevant for our first two research questions **RQ1-2**. It shows that fuzz testing mostly discovers memory errors, and thus can be limited in what properties exactly are covered by the tests, whereas

the Puffin fuzzer is able to also analyze implementation-level protocol faults. In the classification given in Daniele et al. [10], Puffin is a stateful evolutionary fuzzer, since it mutates on a set of known inputs, and adds useful traces to this state. It is able to generate (parts of) a trace as one of its mutations (see Section 4.4), and therefore contains elements of grammar-based fuzzing as well. In terms of knowledge over the SUT, Puffin can be used black-box, but contains a framework for knowledge extraction directly from the SUT. Implementation of this framework requires altering the source code of the SUT, which requires a white-box scenario.

2.2.2 Symbolic verification

Formally verifying a program ensures that it will maintain a certain property (assuming no underlying problems e.g. errors in a dependency). It is a work intensive way of verifying a program. Therefore, it is often only used on small and critical parts of code. Nevertheless, symbolic verification tools have been used for TLS in various ways:

- Cremers et al. [8] used the Tamarin [19] prover to verify the security requirements set in the TLS 1.3 RFC [24] (draft 21 at the time).
- The HMAC-DRBG random generation specification and its implementation in MbedTLS have been proven secure by Ye et al. [36], through the Coq proof assistant.
- Bhargavan et al. [5] built an implementation of TLS in Rust, and then used a framework of different verification tools to prove properties of the program. The framework uses a certain verification tool for its ideal property. For example: ProVerif [6] was used to show message integrity and confidentiality, and F$\star$ to show memory safety. They note that the end product is a competitively performing implementation of TLS, whereas other verified implementations would suffer in performance.
- Sardar et al. [27] applied ProVerif to RA-TLS, a TLS protocol for confidential computing, and found that keys generated through RA-TLS deviated from the TLS specification.

This research shows how properties are proven to hold for real world applications like TLS. As said, this is a work intensive process. Analysis of such a level cannot currently be performed by fuzzers. Puffin is able to find instances where properties of an implementation are broken. These works provide an ideal for fuzzers to reach.

2.2.3 Model learning

Model learning is the process of inferring a state machine that represents a certain system by sending sequences of messages to that system. The resulting state machine represents the inner processes of the implementation, and these could be different than the specification of the program dictates. Various methods can be used to determine if the state machine is correct; from manual inspection to model checking.

Perhaps most related to this work is the research of De Ruiter et al. [26], who used the model learning tool LearnLib to infer state machines of various TLS implementations, three of which showed bugs.

Similar to this is the research of Daniel et al. [9] where the L^* algorithm was used to learn state models for the OpenVPN and the OpenVPN-NL implementations. Daniel et al. split the OpenVPN protocol into three parts: session initialization, TLS handshake and renegotiating session keys. The resulting models mostly show the predicted happy flow, and no significant faults were found.

One of the drawbacks of model learning is that timing sensitive systems can cause non-determinism in recorded test cases. Bruyère et al. [7] presents an extension and implementation of the learning algorithm $L\#$, which allows learning of timed automata. This is then applied to realistic benchmarks, including a model of TCP.

This research shows that model learning is capable of showing faults in the protocol logic of implementations. However, since the methodology uses correctly formed packets, albeit in a strange order, it is not as capable as fuzzing in finding memory management defects.

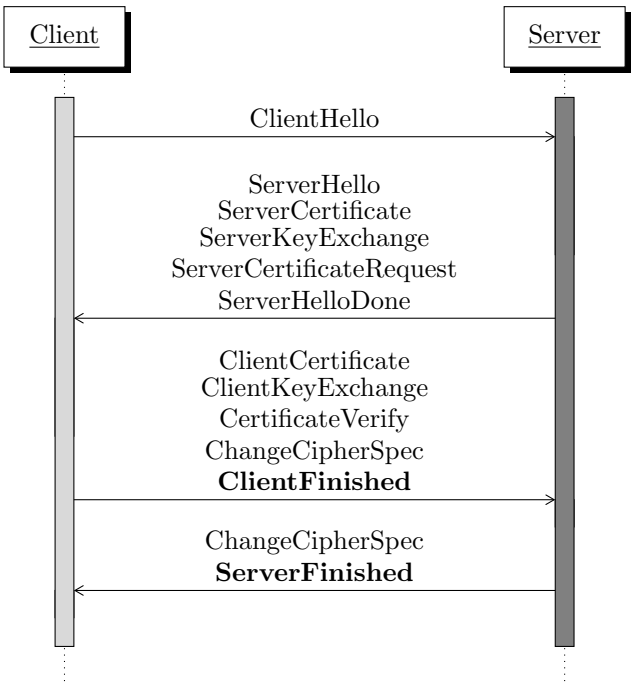

Figure 1: An example happy-flow of the TLS 1.2 handshake

3 Transport Layer Security

In this work we mostly focus on the TLS handshake, as it is the area where negotiations for the session take place, and thus a part where mutation of the communication will be impactful. Additionally, it is the part where we expect a fuzzer with the capabilities of the Dolev-Yao model to have the highest possible impact. In Section 7.4 we expand on this choice.

There are two currently used versions of the TLS protocol: 1.2 and 1.3. Between the two, there are major changes in the handshake, notably TLS 1.3's handshake can be performed in a single round-trip. We now provide a high-level overview of the handshake for TLS 1.2 in Section 3.1 and for TLS 1.3 in Section 3.2. For more detail we refer to RFC 5246 [25] and 8446 [24] for TLS versions 1.2 and 1.3 respectively. We conclude with an overview of the security properties of TLS in Section 3.3.

3.1 TLS 1.2

Figure 1 shows a happy-flow handshake of TLS 1.2, it contains two round trips. First the client sends a *ClientHello* message, whose contents include a random value for replay protection and a list of supported cipher suites: a combination of cryptographic algorithms to be used for various tasks. The server decides on a cipher suite to use from the list the client sent. If using an ephemeral cipher suite, it generates a public & private key pair, and then responds to the client with a set of messages (called a *flight* of messages), which provides information to the client such as: another random value, the chosen cipher suite, the extensions the server supports, the servers public key, and the servers certificate. At this point the client is able to derive a shared secret, called the pre master secret, from the ephemeral server public key it received and its own private key. This derivation is performed according to the cipher suite the server picked. The client responds with their own ephemeral public key, and (if asked for by the server) their own

certificate. After this flight of messages the server is able to derive the same pre master secret the client calculated, using the server private key and client public key. Both agents can now individually combine the pre master secret with the random values sent at the start of the handshake to derive an ephemeral shared master secret. From now on data can be encrypted with the master secret and sent between the hosts, which is signaled by both sides each sending empty *ChangeCipherSpec* messages (effectively an alert to start using encryption), and an encrypted verification message. At this point, the handshake is over and regular application data can be sent.

3.2 TLS 1.3

A happy-flow of the TLS 1.3 handshake is shown in Figure 2. It starts once again with the client sending a *ClientHello* message. This message contains some elements similar to the first two client messages of TLS 1.2: a random value, session ID, list of cipher suites and a generated ephemeral public key. The server chooses a cipher suite from the list the client sent, and generates an ephemeral public key pair. From now on, all the hashes are calculated with the hashing algorithm in the chosen cipher suite.

The server then responds with its own random value and ephemeral public key. From then on, both hosts can derive ephemeral shared secrets according to the cipher suite chosen by the server. This derivation proceeds according to a schedule where at certain points in the handshake the current shared secret is concatenated with a label (specific to that point and client or server) and with the transcript hash of the handshake messages up until now. From this concatenation the next shared secret is then derived. In total eleven different shared secrets are derived and used throughout the rest of the handshake.

Now that the server is able to encrypt messages, it sends an encrypted flight of messages, starting with the *EncryptedExtensions*, to which the client is expected to respond to in later messages. This is followed by a *CertificateRequest*, if the server wants the client to verify themselves with a certificate. Then the server sends their own *ServerCertificate*, followed by a *CertificateVerify* that contains a hash of the handshake messages, signed with the private key associated with the server certificate. The client can verify this signed hash to verify that the server is indeed the owner of the certificate. Finally, the server sends a *ServerFinished* message, which also contains a HMAC of the handshake. Lastly, the client responds with its *ClientCertificate* and *CertificateVerify* messages, in order to authenticate itself to the server. The final handshake message is a *ClientFinished* message.

3.3 TLS security properties

According to the TLS 1.3 RFC [24], the security the protocol provides is focused in three properties:

- **Confidentiality** - TLS provides confidentiality over the data that is sent after the handshake completes. During the handshake a cipher for data encryption is chosen and keys are derived. These are then used to encrypt the data in data messages.
- **Integrity** - In both TLS 1.2 and 1.3 a HMAC over the handshake messages is exchanged through the *ServerFinished* and *ClientFinished* messages. These can be verified by the peer to ensure that no tampering has been performed during the handshake. For application-data messages afterwards, TLS 1.2 either calculates a MAC of the data (and sequence number) in every message, or uses the MAC provided by an AEAD cipher suite if used. TLS 1.3 only supports AEAD cipher

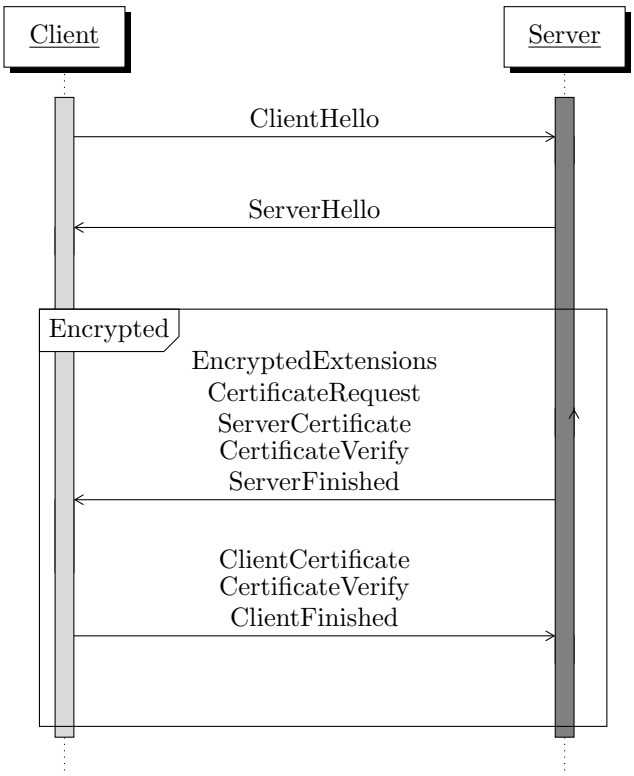

Figure 2: An example happy-flow of the TLS 1.3 handshake

suites, and thus always uses the MAC provided by encryption. Again, the peer can verify these MACs to ensure integrity.

- **Authentication** - During the handshake server and client can request X.509 certificates from each other. This is a chain of signed certificates which ends with a certificate of an authority that the peer trusts. The receiver can verify each entry in the chain with the next entry, ending at the trusted certificate. If this verification succeeds, then the sender is authenticated through the trusted authority to the receiver. Therefore, TLS can provide authentication to its users.

A number of other security properties are given in the RFC, these are related to the three main properties:

- **Session key secrecy & uniqueness** - The keys generated through the handshake should only be known to the client and server. Additionally, they should be unique over unrelated sessions.
- **Downgrade protection** - The negotated cryptographic parameters should not be influenced by a third party.
- **Forward secrecy** - If a key or secret is compromised, an attacker should not be able to decrypt old data.
- **Non-replayability** - An agent should not accept a message that has already been accepted.
- **Length concealment** - The data in a data message can be padded, an attacker should not be able to determine the length of the actual data.

4 Method

In this section, we detail how we intend to answer our research questions. First, we provide a more detailed definition of the two types of defects that we are looking for, as this defines the goal of our experiments (Section 4.1). We then explain how we achieve said goal: starting with a general overview of Puffins architecture (Section 4.2), then we detail the Dolev-Yao model that the tool uses (Section 4.3), from that follows the input mutations Puffin can do (Section 4.4) and the abstraction of TLS we test (Section 4.5). These last two points are the most significant in answering whether Puffin can find implementation-level protocol logic faults, rather than just memory management faults. We finish with our method of verifying that the tool can indeed find these defects: through mutation testing (Section 6.1). This last step provides the answer for our research questions.

4.1 Defect types

So far we have referred to two types of problems: implementation-level protocol logic faults and memory management faults. We now provide definitions for both types, although there is a certain gray area in each. These definitions are adapted from definitions given by Ammann et al. [2] and Swierzy et al. [30].

4.1.1 Memory management faults

Memory management faults are mistakes in the reading or writing of program memory. Most are caused by reading or writing outside the intended memory bounds of a variable and thus accessing other system memory. Common causes are: not enforcing the length of an array or list (memory unsafety), accessing previously freed memory (use-after-free) or mistakes in variable scoping (use-after-return). Note that this does not include "legal" but wrong use of memory: accidentally writing a private key to an output buffer is not a memory management fault (assuming the key fits in the buffer).

4.1.2 Protocol logic faults

For the other type of problem, implementation-level protocol logic faults, we first explain protocol logic faults in general. A protocol provides some properties to its users. For example, as said in Section 3.3, TLS provides authentication of the server to the client and confidentiality of application data sent after the handshake. A fault in protocol logic is then some exploitable defect in either the protocol specification or the implementation logic which breaks one of these properties. Thus, the faults can be separated into two classes: specification-level and implementation-level.

- A specification-level protocol logic defect is simply some weakness in the design of the protocol itself. For example: the state machine of a protocol allows an authentication bypass. It will therefore likely be a problem shared by multiple correct implementations of said protocol. A good example of attacks on a specification-level protocol logic faults are the KRACK attacks on WPA2 [34].

- Implementation-level is, as the name suggests, a mistake in how the protocol is implemented. This can be due to the implementation simply not following the specification. For example, a specification could say that a malformed packet should be dropped immediately. However, in the implementation this malformed packet has a subtle side effect. The result could be an exploitable vulnerability in the program. A well-known example of an implementation-level protocol logic defect is the heartbleed bug [13]. An alternative cause of an implementation-level

defect is when the specification is not detailed enough, or it leaves certain details up to the developer, which are then programmed in a vulnerable way. For example: a client-server protocol might not specify how many clients a single server can handle simultaneously. The developer might limit this to, say, two. However, if a third client attempts to connect to the server, it skips some authentication step. This is a fault in the protocol logic, however, the problem is mainly caused by the implementation of the protocol and possibly the configuration of the server. In the case of configuration mistakes, we consider them implementation-level defects if the value is configured within a correct bound set by the specification.

There is a gray-area within the alternative cause: You could say that a problem caused by a developer filling in the blanks of an incomplete specification is, at its root, a problem with the specification lacking information, and thus, a specification-level issue rather than an implementation-level issue. We consider these situations a combination of both. As such, Puffin should be able to find these faults, as they are also an implementation-level protocol logic fault. Theoretically, Puffin is capable of finding purely specification-level protocol logic faults through its security oracle (defined in Section 4.3.4), this would require the security oracle to have stronger checks than the protocol itself. We consider this potential out of scope for this work, and refer to Future work in Section 9.4.

Note that an implementation-level protocol logic fault can be caused by a vulnerability in memory management. Continuing with our earlier maximum-two-connections example: if the third client connecting would cause a buffer overflow instead of an authentication skip, there is a clear overlap in the two types of defects. During our verification in Section 6.1.3, we consider the four different combinations in this overlap (one of the types, both and none).

4.2 Puffin fuzzing loop

We now provide a high level description of the Puffin fuzzer, several concepts behind the program are explained further in the following sections.

Puffin by Ammann et al. [2] is adapted from LibAFL, originally by Fioraldi et al.[14]. As such, the general loop the program executes is similar. LibAFL is a mutation based fuzzer, where an execution starts with a set of input traces, called the **corpus**, which are mutated and ran against the SUT. Various observations can be made during execution, and with those the fuzzer determines if a trace is *interesting*, an *objective*, or neither. In the case of Puffin, a trace is **interesting** if its execution has covered new edges in the SUT. Coverage is tracked through the sanitizer coverage tool, part of LLVM[3], so an increase in coverage means the input reached a new edge in the control flow graph of the SUT code. Every interesting new trace is added to the corpus. An **objective** trace accomplishes the goal of the fuzzer, in our case this is either a runtime crash, a timeout (which is also deemed a runtime crash), or when the security oracle finds that a security property is broken (see Section 4.3.4). The objectives are recorded separately, and are not added to the corpus. A trace that is not interesting and not an objective is dropped altogether. Although one could argue that storage of uninteresting traces would prevent testing the same input twice (or more), the quantity of traces and lookup time for duplicates would grow too fast to manage. When all the traces in the starting corpus have been executed once, the fuzzing loop starts.

Figure 3 shows a flowchart of the fuzzing loop in Puffin, specifically the TLS version: TLSPuffin, although the structure will be similar for other protocols. The loop begins when the *scheduler* selects a trace from the *corpus*, which initially contains only seed

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

traces. The traces are specified as term in the Dolev-Yao model, (see Section 4.3), and are thus symbolic at this point. This trace is then mutated by the *mutator* (see Section 4.4). The mutations are applied according to an optimized strategy [14], and they each contain some elements of randomization. A single trace is mutated multiple times with different mutations. This new trace is then sent to the *mapper*, which can translate the symbols of the trace to concrete TLS packets. These packets are then sent one-by-one to the *harness*. The harness maintains one or more active instances of the *SUT*, and executes the byte packets received from the mapper on the SUT. During execution observations are made by the *observer*, and claim information is gathered by the *claimer* (see Section 5.3). The *feedback* oracle uses the information from the observation to determine whether the execution was interesting. If so, the trace is added to the corpus. Finally, the *objective oracle* determines whether the execution achieved an objective. For Puffin, this can be done through either the *memory oracle*, which recognizes crashes and timeouts from information of the observer, or the *security oracle*, which uses information from the claimer to decide whether a security property did not hold during execution (see Section 4.3.4). The memory oracle itself is a simple detection whether the SUT crashed or timed out. This is assisted by instrumentalizing the SUT with LLVMs AddressSanitizer[4] and (later in our experiments) UndefinedBehaviorSanitizer[5]. These tools can be configured to crash the SUT when memory mismanagement or undefined behaviour are detected, the memory oracle will detect these crashes. If either of the objective oracles concludes that an objective was achieved, then the mutated trace is added to the *objectives*. Then the loop restarts, and a new trace is picked from the corpus.

It is this security oracle that allows Puffin to discover errors and unintended behaviour which a fuzzer that only checks for memory management would miss. When working with Puffin, a user can specify security properties and when they should hold. These properties represent a certain measure of trust in information the SUT has sent and received throughout execution. If a property is broken, it means the SUT trusted some information when it should not have. For example, a SUT could believe some encryption key was agreed upon, whilst it was in fact changed by the attacker. In other words, this system of security properties and oracle allows Puffin to find protocol implementation errors as defined in Section 4.1.2. This specific problem would not be caught by a fuzzer that only checks for memory errors, and would therefore go unreported. As such, this system is directly related to our research questions **RQ1-2**.

The other parts that Puffin adds on LibAFL: DY traces and related mutations, use the knowledge of a protocol that the DY model provides to build traces that are close to correct. Since inputs are based on either a happy-path trace or a trace that discovered new edges in the SUT, the probability that a mutated input is a legal trace of the protocol is higher than when more conventional means of mutation are used.

4.3 Dolev-Yao model

The Dolev-Yao model as defined by Dolev et al. [11] is a symbolic model wherein properties of cryptographic protocols can be proven. The model assumes that there is an active saboteur as a man-in-the-middle between two honest agents. To define Dolev-Yao (DY) terms and to reason about traces and security properties, we use the notation of the DY model as given in Ammann et al. [2] and add a framework for an extraction function, which was not clearly explained in the original work. Ammanns notation was adapted from the applied π-calculus, a process algebra that is also used in the ProVerif verification tool [6]. The DY-model is used by Puffin, which transforms DY-terms to bit-messages and sends them to systems under testing. During execution the fuzzer

[4] `https://clang.llvm.org/docs/AddressSanitizer.html`
[5] `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

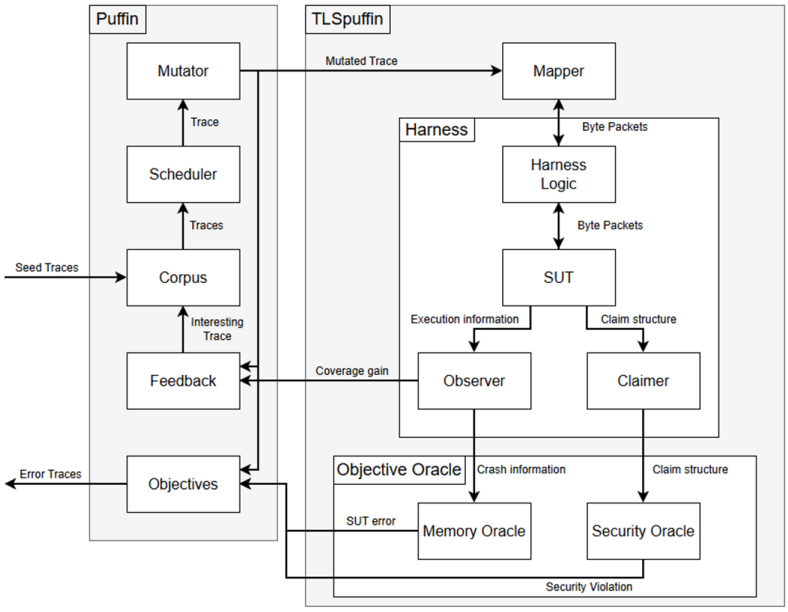

Figure 3: An overview of the TLSPuffin achitecture

creates security claims over the trace, which also uses some of the DY-model notation. In this section, we show the formal structure of the DY-model by explaining, in order, the notation of terms (Section 4.3.1), then the notation of traces (Section 4.3.2), trace semantics (Section 4.3.3), and finally the idea behind security claims (Section 4.3.4). After every section we provide a running example to clarify further.

4.3.1 Term algebra syntax

A Dolev-Yao model consists of a set of traces, which are built from terms. Each term can be translated to a message (see Section 4.3.3). Syntactically, terms are built from the atomic units *operators* and *variables*: An operator intuitively can be seen as a "function" with a certain arity. Operators with the arity of zero are called *atomics*, which can be seen as constant values (although they need not necessarily be a set value, but they should be independent from the value of other terms, like nonces). Variables are then elements of named information that will be replaced with concrete terms during execution of the model. Formally: We define the set of **operators** $\mathcal{O}$ and the arity function $arity : \mathcal{O} \to \mathbb{N}$. The **signature** $\Sigma = \{(o, n) \mid o \in \mathcal{O}, n = arity(o)\}$ is the set of operators with their associated arities, where a set of **atomics** $\mathcal{A}$ is a special set of operators with the arity of zero. The set $\mathcal{V}$ is a countably infinite set of **variables**. This makes the set of **terms** $\mathcal{T}$ the set of variables, atomic operators, and the terms resulting from applying non-atomic operators to other terms. This is inductively defined in Figure 4.

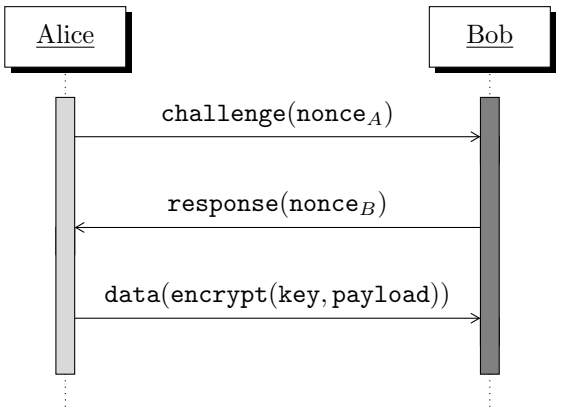

Figure 5: An example key-negotiation protocol

$$
\begin{array}{rll}
\tau \in \mathcal{T} ::= & v \mid & \text{(where } v \in \mathcal{V}\text{)} \\
& a \mid & \text{(where } a \in \mathcal{A}\text{)} \\
& o(\tau'_1, ..., \tau'_n) & \text{(where } (o, n) \in \Sigma \text{ and } \tau'_i \in \mathcal{T} \text{ for } 1 \leq i \leq n\text{)}
\end{array}
$$

Figure 4: Definition of terms

Running example To illustrate the DY-model, we consider a simple key negotiation protocol shown in Figure 5. In the protocol, Alice sends a nonce to Bob, who responds with their own nonce. Both then generate the same cryptographic key using both numbers. Alice finally encrypts a payload of data with that key, and transmits it to Bob. The exact methods of key derivation, encryption and decryption are left abstract, since the example serves to illustrate the term-construction, information extraction and trace mutating, and for this goal only the notion of a key and encrypt function suffice. Figure 6 shows sets of structural elements for DY-terms corresponding with this protocol. The set of operators (shown in line 1 of Figure 6) is combined from the function- names, packet names, and named data. The set of atomics (line 2) consists of the operators with arity zero. The list of variables $\mathcal{V}$ is infinite, but for the remainder of this chapter we specify a number of obvious choices and show these in line 3. The complete signature on line 4 contains all the operators with their arity. We've shortened the list of atomics for brevity. From the signature we can build the set of terms. We show a small fragment of the infinite set on line 5.

4.3.2 DY-trace syntax

A trace in the Dolev-Yao model is a sequence of messages where entities (called agents) communicate with a single attacker. Thus we define a finite set of **channels** $\mathcal{C}$ where each channel $c \in \mathcal{C}$ represents the communication between a particular agent and the attacker. For every message we then specify whether it has been sent by the attacker into the agent (`in`), or whether the agent has output the message into the channel (`out`). So, in terms of function naming, we communicate from the viewpoint of the agents. A **DY trace** is thus defined as a sequence of **actions** $a_1 \cdots a_n$ where every action a_i for $1 \leq i \leq n$ is either:

- $\texttt{out}(c, x)$ where $(c \in \mathcal{C}, x \in \mathcal{V})$.

$$\mathcal{O} = \{\texttt{challenge}, \texttt{nonce}_A, \texttt{response}, \texttt{nonce}_B, \texttt{data}, \texttt{encrypt}, \texttt{key}, \texttt{payload}\} \quad (1)$$

$$\mathcal{A} = \{\texttt{nonce}_A, \texttt{nonce}_B, \texttt{key}, \texttt{payload}\} \quad (2)$$

$$\mathcal{V} = \{x, x_{\texttt{chall}}, x_{\texttt{full}}, x_{d_1}, x_{\texttt{nonce}A}, y, y_{\texttt{resp}}, y_{\texttt{full}}, y_{d_2}, z_{\texttt{data}}, z_{\texttt{key}}, z_{\texttt{pl}}, \dots\} \quad (3)$$

$$\Sigma = \{(a, 0) \mid \forall a \in \mathcal{A}\} \cup \{(\texttt{challenge}, 1), (\texttt{response}, 1), (\texttt{data}, 1), (\texttt{encrypt}, 2)\} \quad (4)$$

$$\mathcal{T} = \{x_{\texttt{chall}}, x_{\texttt{nonce}A}, \dots, z_{\texttt{pl}}, \texttt{nonce}_A, \texttt{nonce}_B, \texttt{data}, \texttt{challenge}(\texttt{nonce}_A), \texttt{challenge}(\texttt{nonce}_B), \texttt{challenge}(\texttt{payload}), \texttt{data}(\texttt{nonce}_A), \dots\} \quad (5)$$

Figure 6: DY-model structures for example protocol

- $\texttt{in}(c, t)$ where $(c \in \mathcal{C}, t \in \mathcal{T})$.

Note that the attacker receives variables and sends terms: Before term τ can be sent with $\texttt{in}(c, \tau)$, any variable present in τ has to be substituted with atoms, operators that do not contain variables, or concrete data (see Section 4.3.3). These terms can either be generated by the attacker themselves, or obtained from messages received through $\texttt{out}(c', x)$ from any $c' \in \mathcal{C}$. Later, in Section 4.3.3 we can obtain "deeper" information through pattern matching on a received variable: If variable x matches with a pattern $\texttt{encrypt}(k, d)$, we can use k in a future message that requires a public key. This sequence structure of in- and output resembles the applied π-calculus used in ProVerif [6], although some concepts from that model like parallel execution are missing in our DY-traces.

Example trace We continue with our key-negotiation example, where Alice and Bob send messages into a channel which an attacker can control: Figure 7 shows the flow of messages. Compared to the earlier Figure 5, we replace the terms that Alice outputs with a variable like $x_{\texttt{chall}}$ which represents the message. Since, as said, the attacker receives variables. Additionally, in the terms that the attacker sends we replace some operators with variables, to show where extracted information (see Section 4.3.3 upcoming) can be substituted in.

To move from the graphical representation of the happy-flow to a DY trace, the agents now get their own channels: $\mathcal{C} = \{c_A, c_B\}$. The DY-trace is then built from `in` and `out` actions, starting with Alice who sends the first nonce to the attacker, who then passes it through to Bob: An example trace of the happy-flow of our protocol is given in Figure 8. The man-in-the-middle setting that the DY-model assumes can clearly be seen. Note that this is not the only possible trace for the happy-flow, in this specific example we assume the attacker has insight into the `encrypt` operator, and is able to find a key $z_{\texttt{key}}$ to use over $z_{\texttt{payload}}$. Otherwise they would not be able to complete the last `in` action correctly. A different instantiation would be for the last `in` to use $\texttt{data}(z_{\texttt{data}})$ instead. Here the attacker simply passes along the encrypted information given to them by A. A trace with this call would still succeed even without our earlier assumption. However, it is plain to see that there should be a bigger potential for our desired application, fuzz testing, when the encrypt call is exposed.

4.3.3 DY-trace semantics

In order to concretize the output terms of a trace into a bitstring that can be sent to an agent in the system, we need a semantics of DY-traces. Ammann et al. [2] built

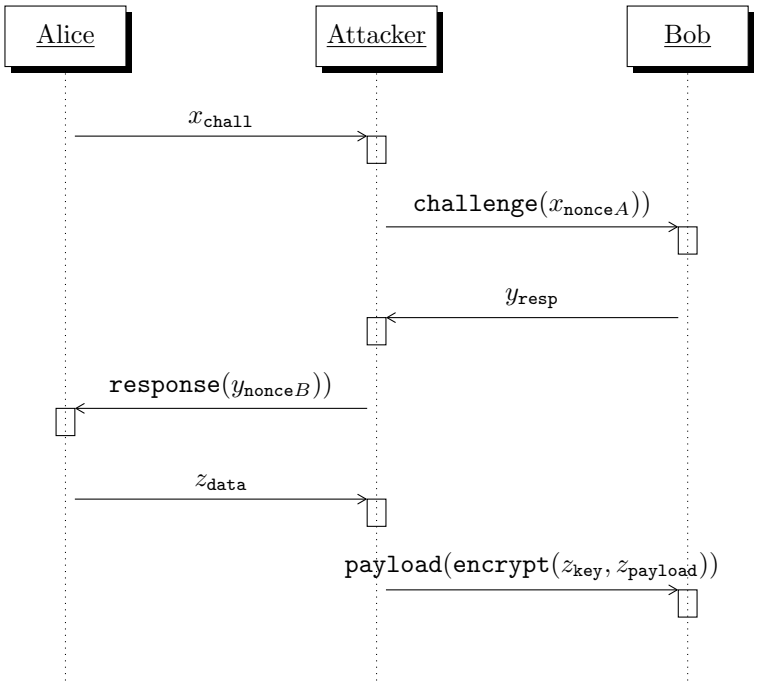

Figure 7: Example protocol with a man-in-the-middle

$$\texttt{out}(c_A, x_{\texttt{chall}}) \cdot \texttt{in}(c_B, \texttt{challenge}(x_{\texttt{nonce}A})) \cdot$$
$$\texttt{out}(c_B, y_{\texttt{resp}}) \cdot \texttt{in}(c_A, \texttt{response}(y_{\texttt{nonce}B})) \cdot$$
$$\texttt{out}(c_A, z_{\texttt{data}}) \cdot \texttt{in}(c_B, \texttt{payload}(\texttt{encrypt}(z_{\texttt{key}}, z_{\texttt{payload}})))$$

Figure 8: Example happy-flow trace for key-negotiation protocol

a generic framework which can additionally be extended to a formal language like π-calculus. The semantics of DY-traces are intuitively structured as follows: States are associated with every channel in the trace, "client"-states so to speak. The attacker is provided a separate, initially empty, attacker state. Executions of the `in` and `out` actions in the trace will then update the state of their associated channels (for example: $\texttt{in}(c_1, x)$ will update the state for c_1). The attacker state meanwhile stores every message received from the clients via `out`. During execution, the attacker can use variables stored in its attacker state to construct new messages, which can then be sent through `in`.

States The first element in this process we discuss is the notion of states. We define the set of channel **states** $\mathcal{S}$, from which every channel c in a trace is given a state s_c. A single global client state is a function $s : \mathcal{C} \to \mathcal{S}$, that returns the state of any given channel. The set of initial states $\mathcal{S}_0$ is a subset of $\mathcal{S}$ before running any elements of a trace. A global state is initial if it only returns initial channel states. In the semantic-framework these states will be updated for every `in` and `out` processed. The attacker has their own type of state: ϕ, which will store information from received messages. Therefore, we define the set of **messages** $\mathcal{M}$ and the **extraction** function ϵ, which extracts knowledge from a message. It is important to note that extraction cannot always be performed: If a message was constructed from a term with a hash function $\texttt{h}(\tau)$, we are not able to extract τ. Thus, we need the ability to represent unextractable data as-is. For this purpose we define data $\mathcal{D}$, which will be highly implementation specific and therefore kept abstract (later in this work, data will be in the form of bitstrings). In conclusion, this makes $\mathcal{M}$ structurally similar to terms $\mathcal{T}$: A message consists of either an atomic operator, an operator performed over sub-messages, or concrete data. M is defined in Figure 9. The extraction function $\epsilon : \mathcal{V} \times \mathcal{M} \to \mathcal{V} \times \mathcal{M}$ is also implementation specific, and has to be implemented separately (see paragraph "example concretization and extraction"). It should bind variables to pieces of extracted knowledge, in order to add them to the attacker state ϕ. Note that the added variables do not have to be fresh. This allows a targeted concretization during message construction later. However, selecting variables in this way requires domain knowledge, and is therefore implementation specific. With its ingredients present, the attacker state $\phi : \mathcal{V} \to \mathcal{M}$ is then defined as a storage that binds variables to extracted knowledge. It will throughout execution be updated via the extraction function. The set of attacker states is Φ.

$$
\begin{aligned}
m \in \mathcal{M} \ ::= \ & d \mid && \text{(where } d \in \mathcal{D}\text{)} \\
& a \mid && \text{(where } a \in \mathcal{A}\text{)} \\
& o(m'_1, ..., m'_n) && \text{(where } (o, n) \in \Sigma \text{ and } m'_i \in /mathcalM \text{ for } 1 \leq i \leq n\text{)}
\end{aligned}
$$

Figure 9: Definition of messages

Example messages As messages are structured similarly to terms, our example set of messages shown in Figure 10 is also similar to the terms shown in line 5 of Figure 6. The example messages are focused around the last part of our protocol, the data packet, in order to show the increasing number of operators from the inductive definition of the structure. Note that, with our running example, we would have provided some examples of state here, however we find that our example of trace execution at the end of this Section 4.3.3 also provides insight into states, therefore we refer to that example. Similarly, we have defined an example framework for information extraction. However, our example uses the interpretation function `inter`, which is defined in the

$$
\begin{aligned}
\mathcal{M} = \{ & d_1, d_2, d_3, d_4, \ \texttt{key}, \texttt{payload}, \\
& \texttt{encrypt}(\texttt{key}, \texttt{payload}), \texttt{data}(\texttt{encrypt}(\texttt{key}, \texttt{payload})), \\
& \texttt{encrypt}(d_1, d_1), \texttt{encrypt}(d_1, d_2), \ldots, \texttt{encrypt}(d_1, \texttt{payload}), \\
& \texttt{encrypt}(d_2, d_1), \ldots, \texttt{encrypt}(\texttt{payload}, \texttt{payload}), \\
& \texttt{data}(d_1), \ldots, \texttt{data}(\texttt{payload}), \\
& \texttt{data}(\texttt{encrypt}(d_1, d_1)), \ldots, \texttt{data}(\texttt{encrypt}(\texttt{payload}, \texttt{payload})), \ldots \}
\end{aligned}
$$

Figure 10: Example set of messages

$$
\begin{aligned}
\sigma_\phi(v) &= \phi(v) && \text{where } v \in \mathcal{V} \\
\sigma_\phi(o(\tau_1, ..., \tau_n)) &= o(\sigma(\tau_1), ..., \sigma(\tau_n))
\end{aligned}
$$

Figure 11: The substitution function

following paragraph "Messages", therefore, we provide the framework at the end of that paragraph.

Messages We move on to message construction. Intuitively, a message is created from a term through two processes: **concretization** and **interpretation**. We explain these separately, although they can be merged into a single final function. Concretization is the process of removing the variables in a term. This is done by exchanging them with operators or atoms. To do this, we define a **substitution** function $\sigma_\phi : \mathcal{T} \to \mathcal{M}$, which uses information from the agent state ϕ. It therefore constructs a message from a term, since both structures contain operators and atoms, and the term-exclusive variables are replaced with messages. The function is shown in Figure 11.

The concrete representation of a message is then created through **interpretation** functions $[\![\cdot]\!]$, which are defined for every operator $o \in \mathcal{O}$, taking the arity of o into account: $[\![o]\!] : \mathcal{D}^i \to \mathcal{D}$ where $(o, i) \in \Sigma$. This notation is similar to a Sigma-algebra [12]. Then we use a single function $\texttt{inter} : \mathcal{M} \to \mathcal{D}$ to pass through the message. The resulting function is shown in Figure 12. These $[\![\cdot]\!]$ functions are protocol-specific, since they require knowledge about the concrete representation of terms, and therefore have to be implemented separately. As said, we can combine the concretization function σ and interpretation function $\texttt{inter}$ into a single function. This new interpretation function requires an associated agent state $\phi \in \Phi$ and is defined as $[\![\cdot]\!]_\phi : \mathcal{T} \to \mathcal{D}$. Figure 13 shows the combined function.

$$
\begin{aligned}
\texttt{inter}(o(\tau_1, \ldots, \tau_n)) &= [\![o]\!](\texttt{inter}(\tau_1), \ldots, \texttt{inter}(\tau_n)) && \text{if } (o, n) \in \Sigma \\
\texttt{inter}(a) &= [\![a]\!] && \text{if } a \in \mathcal{A} \\
\texttt{inter}(d) &= d && \text{if } d \in \mathcal{D}
\end{aligned}
$$

Figure 12: The interpretation function

$$[\![\tau]\!]_\phi = \texttt{inter}(\sigma_\phi(\tau))$$

Figure 13: The combined concretization and interpretation function

$$\texttt{extr}(d_a) = \{d_a, a\} \text{ for } a \in \mathcal{A} \text{ if } [\![a]\!] = d_a \tag{6}$$

$$\texttt{extr}(d_o) = \{d_o\} \cup (\bigcup_{i=1}^{n} \texttt{extr}(d_i)) \cup \tag{7}$$
$$\{o(e_1, \ldots, e_n) \mid \forall_{j=1}^{n}[e_j \in \texttt{extr}(d_j)]\}$$
$$\text{where } (o, n) \in \Sigma \text{ such that } [\![o]\!](d_1, \ldots, d_n) = d_o$$

$$\texttt{extr}(d) = \{d\} \qquad \text{otherwise} \tag{8}$$

Figure 14: Example extraction function

Example concretization and extraction Now we define a framework for information extraction, which mostly reduces the task into calls to interpretation functions $[\![\cdot]\!]$. The extraction function, defined earlier as $\epsilon : \mathcal{V} \times \mathcal{M} \to \mathcal{V} \times \mathcal{M}$ will extract knowledge from data and then bind variables to that knowledge. We split these two tasks by defining the helper function $\texttt{extr} : \mathcal{M} \to 2^{\mathcal{M}}$ which will be used for the first step, extracting knowledge from data. The main extraction function ϵ will then perform the binding of variables. Therefore we start with defining the helper extraction function `extr`. Figure 14 shows a definition. In it, we match concrete data returned by the interpretation function $[\![\cdot]\!]$ with the data given to the extraction function. Our extraction has to return every item of knowledge found, not only the entire structure of the original term, therefore, it is not only the inverse of `inter`. The function has two base cases:

- First, line 6: If the data provided is equal to what the interpretation function returns for atomic data, then the original data was an atomic operator pointing to that data. Therefore, the result is a set containing the data given, and the atomic operator it matches with.
- The second base case on line 8 can be seen as a failure case; if the given data cannot be matched through the interpretation, then we return just that data. This case is necessary for operators that modify data in such a way that the original cannot be retrieved. The prime example is a hash function, but it could also occur through encryption, for example if our system under test is a black-box and we deem any type of key derivation out of scope.

The inductive step on line 7 assumes the data d_o originates from an operator applied to some arguments, and therefore we look for a possible `inter` call that results in exactly the data we're analyzing now. Since the `inter` function can use concrete data as arguments for an operator, we can effectively rebuild the term in a top-down manner through this induction. Once we find the correct message $o(d_1, \ldots, d_n)$ we also attempt to extract information from $d_1, \ldots, d_n$. The result of a single inductive step is the original data, the extracted function with all it's arguments as *either* extracted information *or* concrete data, and this information and data lifted from the function. This to ensure that we return every item of knowledge found. The second part of extracting information is binding the found information to variables. An example implementation of an extraction function ϵ is given in line 9 of Figure 15. In it, we bind the full packet to the given

$$\epsilon(v,d) = \{v,d\} \cup \{(w,m) \in \mathcal{V} \times \texttt{extr}(d) \mid m \neq d\} \tag{9}$$

$$\epsilon'(v,d) = \begin{cases} (y^*, m) & \text{if } m = \texttt{encrypt}(\texttt{key}, \texttt{payload}) \text{ in } (w,m) \in \epsilon(v,d) \\ (v,m) \in \epsilon(v,d) & \text{otherwise} \end{cases} \tag{10}$$

Figure 15: Example extraction variable assignment

$$\begin{aligned} \epsilon(v,d_4) = \{&(v,d_4), (w, \texttt{data}(d_3)), (w_{\texttt{data}}, d_3), \\ &(x, \texttt{data}(\texttt{encrypt}(d_1,d_2))), (x_{\texttt{key}}, \texttt{data}(\texttt{encrypt}(\texttt{key},d_2))), \\ &(x_{\texttt{pl}}, \texttt{data}(\texttt{encrypt}(d_1,\texttt{payload}))), \\ &(x_{\texttt{both}}, \texttt{data}(\texttt{encrypt}(\texttt{key},\texttt{payload}))), \\ &(y, \texttt{encrypt}(d_1,d_2)), (y_{\texttt{key}}, \texttt{encrypt}(\texttt{key},d_2)), \\ &(y_{\texttt{pl}}, \texttt{encrypt}(d_1,\texttt{payload})), \\ &(y_{\texttt{both}}, \texttt{encrypt}(\texttt{key},\texttt{payload})), \\ &(z_1,d_1), (z_2,d_2), (z_{\texttt{key}}, \texttt{key}), (z_{\texttt{pl}}, \texttt{payload})\} \\ \text{where } &\texttt{inter}(\texttt{data}(d_3)) = d_4, \texttt{inter}(\texttt{encrypt}(d_1,d_2)) = d_3, \\ &\texttt{inter}(\texttt{payload}) = d_2, \texttt{inter}(\texttt{key}) = d_1 \end{aligned}$$

Figure 16: Example extraction result

variable, which is a consequence of $\texttt{out}(c,x)$ binding a message to x independently from extraction, although both results have to update the same state. Therefore, we allow this binding of the original packet. The right hand side of the union simply maps variables to knowledge. Then, by adding extra clauses to the function, we can specify that certain pieces of knowledge need to be bound to specific variables. This lets us target certain knowledge for reuse in a trace. In line 10 for example, we enforce that if the message $\texttt{encrypt}(\texttt{key},\texttt{payload})$ is present, it should be assigned to the variable y^*. This y^* can then be used in the term of an $\texttt{in}(c,t)$ action in a trace, which would be substituted with the desired information during execution. In Figure 16 we show the result of applying ϵ to a message d_4, which contains the representation of a correct `data` packet in our earlier defined key-negotiation protocol. Variables are clearly bound to internal items of knowledge.

Updating state The final ingredients in executing a DY-trace are functions that actually execute received messages and return other messages, in other words, functions that change the *state* of both attacker and channel. The functions `input` and `output` will do just that. Notably these functions are different from earlier defined `in` and `out`. Those are parts of the DY-trace and provide terms to be sent as well as namespaces for returned variables. They do not interact with states at all. The function $\texttt{input} : \mathcal{S} \times \mathcal{M} \to \mathcal{S}$ receives and executes a message in a certain state of a program, and returns the updated state of the program. Its counterpart $\texttt{output} : \mathcal{S} \to \mathcal{S} \times \mathcal{M}$ represents an incoming message from the attackers point of view. It also updates the state of a channel, since a program could simply send multiple different messages in a row without outside input.

Trace execution With every puzzle piece on the board we can finally show how a DY-trace is executed. It is a simple system of two transition rules, one for `in` & `input` and one for `out` & `output`. Both will continuously update the global state, while the `out` rule additionally updates the attacker state with extracted knowledge. Formally: For a DY-trace $a_1 \cdot \dots \cdot a_n$ where every a_i is either $\texttt{in}(c, t)$ or $\texttt{out}(c, x)$. Given a global initial state s and an empty attacker state ϕ, an **execution** proceeds as shown in Figure 17:

$$(s, \phi) \xrightarrow{\texttt{out}(c,x)} (s[c \mapsto s'_c], \phi \cup \epsilon(x, m)) \quad \text{where } \texttt{output}(s(c)) = (s'_c, m) \tag{11}$$

$$(s, \phi) \xrightarrow{\texttt{in}(c,t)} (s[c \mapsto s'_c], \phi) \quad \text{where } \texttt{input}(s(c), [\![t]\!]_\phi) = s'_c \tag{12}$$

Figure 17: Transition rules for DY-trace execution

Example execution Figure 18 shows how the states of channels and attacker change during execution of our happy-flow trace from Figure 8. It begins with a starting state s and empty ϕ, and applies the transition rules in an alternating way. Each `in` performs an interpretation, and each `out` performs extraction. The end result is a pair where both channels' states have been updated thrice, and the attacker state contains information from the three received packets. The results of extraction ϵ are shown separately in Figure 19. We've chosen to not name some of the variables for visual clarity, as they are unused during the execution.

In conclusion This section has shown a framework to execute a DY-trace. In order to customize the behavior of a DY-model, two functions have to be implemented: The interpretation function $[\![\cdot]\!]_\phi$ and the extraction function ϵ. In the rest of this work, we use a practical implementation that interprets bitstrings.

4.3.4 Security properties

A **security property** in the fuzzer of Ammann et al. [2] is expressed as a **claim** during execution of a trace. One claim is structured like a predicate over messages: $\texttt{c}(m_1, \dots, m_n)$ where `c` is the name of the claim, and $m_i \in \mathcal{M}$ for $1 \leq i \leq n$ are messages. The set of all claims is defined as $\mathbb{C}$. A claim is made by a specific agent during the trace. These properties are, again, protocol- dependent. This is not only the case for *what* we claim, but also *when* we say a claim is satisfied. We define the function that returns all claims made by an agent in a given local state: $\texttt{claims} : \mathcal{S} \to \mathbb{C}$. This function can be combined with a DY-execution $(s_0, \phi) \to \dots \to (s_n, \phi_n)$, to create a **trace of claims** C as follows: $C = C_0 \cdot \dots \cdot C_n$ where $C_i = \bigcup_{c \in \mathcal{C}} \texttt{claims}(s_i(c))$. This shows the construction of the set of claims over time, and allows us to reason with a timing aspect. To achieve this, we extend a claim with positioning: $c(m_1, \dots, m_n) \mathbin{@} i$, where i is an index in the trace of claims. This positioned claim is true if the normal claim is true at position i in the execution. Now we are able to build logical expressions with positioned claims (see further for an example). This framework of claims allows us to research whether a property holds in a trace.

To construct an example of security properties we refer back to our running example of Sections 4.3.1 and 4.3.2. First, we define $\texttt{Ag}(m_1, m_2, m_d)$ as the **agreement**-claim; it is created when an agent, who is in some way identified by message m_1, believes to have agreed with an agent identified by m_2 over the data of message m_d. For our example protocol, identification through a message can be achieved by associating a

$$
\begin{array}{r}
(s, \phi = \{\}) \\
\Big\downarrow \ \mathtt{out}(c_A, x) \quad \text{where } \mathtt{output}(s(c_A)) = (s'_A, m_{\mathtt{chall}}) \\
(s[c_A \mapsto s'_A], \phi = \{\} \cup \epsilon(x, m_{\mathtt{chall}})) \\
\Big\downarrow \ \mathtt{in}(c_B, \mathtt{challenge}(x_{\mathtt{nonce}A})) \quad \text{where } \mathtt{input}(s(c_B), [\![\mathtt{challenge}(x_{\mathtt{nonce}A})]\!]_\phi) = s'_B \\
(s[c_A \mapsto s'_A, c_B \mapsto s'_B], \phi = \epsilon(x, m_{\mathtt{chall}})) \\
\Big\downarrow \ \mathtt{out}(c_B, y) \quad \text{where } \mathtt{output}(s(c_B)) = (s''_B, m_{\mathtt{resp}}) \\
(s[c_A \mapsto s'_A, c_B \mapsto s''_B], \phi = \epsilon(x, m_{\mathtt{chall}}) \cup \epsilon(y, m_{\mathtt{resp}})) \\
\Big\downarrow \ \mathtt{in}(c_A, \mathtt{response}(y_{\mathtt{nonce}B})) \quad \text{where } \mathtt{input}(s(c_A), [\![\mathtt{response}(y_{\mathtt{nonce}B})]\!]_\phi) = s''_A \\
(s[c_A \mapsto s''_A, c_B \mapsto s''_B], \phi = \epsilon(x, m_{\mathtt{chall}}) \cup \epsilon(y, m_{\mathtt{resp}})) \\
\Big\downarrow \ \mathtt{out}(c_A, z) \quad \text{where } \mathtt{output}(s(c_A)) = (s'''_A, m_{\mathtt{data}}) \\
(s[c_A \mapsto s'''_A, c_B \mapsto s''_B], \phi = \epsilon(x, m_{\mathtt{chall}}) \cup \epsilon(y, m_{\mathtt{resp}}) \cup \epsilon(z, m_{\mathtt{data}})) \\
\Big\downarrow \ \mathtt{in}(c_B, \mathtt{data}(z_{\mathtt{key}}, z_{\mathtt{pl}})) \quad \text{where } \mathtt{input}(s(c_B), [\![\mathtt{data}(\mathtt{encrypt}(z_{\mathtt{key}}, z_{\mathtt{pl}}))]\!]_\phi) = s'''_B \\
(s[c_A \mapsto s'''_A, c_B \mapsto s'''_B], \phi = \epsilon(x, m_{\mathtt{chall}}) \cup \epsilon(y, m_{\mathtt{resp}}) \cup \epsilon(z, m_{\mathtt{data}}))
\end{array}
$$

Figure 18: Example execution

$$
\begin{aligned}
\epsilon(x, m_{\mathtt{chall}}) = &\{(x, m_{\mathtt{chall}}), (x_{\mathtt{chall}}, \mathtt{challenge}(d_1)), (x_{\mathtt{full}}, \mathtt{challenge}(\mathtt{nonce}_A)), \\
&(x_{d_1}, d_1), (x_{\mathtt{nonce}A}, \mathtt{nonce}_A)\} \\
\epsilon(y, m_{\mathtt{resp}}) = &\{(y, m_{\mathtt{resp}}), (y_{\mathtt{resp}}, \mathtt{response}(d_2)), (y_{\mathtt{full}}, \mathtt{response}(\mathtt{nonce}_B)), \\
&(y_{d_2}, d_2), (y_{\mathtt{nonce}B}, \mathtt{nonce}_B)\} \\
\epsilon(z, m_{\mathtt{data}}) = &\{(z, m_{\mathtt{data}}), (\_, d_{\mathtt{enc}}), (\_, d_{\mathtt{key}}), (z_{\mathtt{key}}, \mathtt{key}), (\_, d_{\mathtt{pl}}), (z_{\mathtt{pl}}, \mathtt{payload}), \\
&(\_, \mathtt{encrypt}(d_{\mathtt{key}}, d_{\mathtt{pl}})), (\_, \mathtt{encrypt}(\mathtt{key}, d_{\mathtt{pl}})), \\
&(\_, \mathtt{encrypt}(d_{\mathtt{key}}, \mathtt{payload})), (\_, \mathtt{encrypt}(\mathtt{key}, \mathtt{payload})), \\
&(\_, \mathtt{data}, (\mathtt{encrypt}(d_{\mathtt{key}}, d_{\mathtt{pl}}))), (\_, \mathtt{data}(\mathtt{encrypt}(\mathtt{key}, d_{\mathtt{pl}}))), \\
&(\_, \mathtt{data}(\mathtt{encrypt}(d_{\mathtt{key}}, \mathtt{payload}))), \\
&(\_, \mathtt{data}(\mathtt{encrypt}(\mathtt{key}, \mathtt{payload}))), (\_, \mathtt{data}(d_{\mathtt{encrypt}}))\}
\end{aligned}
$$

Figure 19: Extraction results for example execution

$$
\begin{aligned}
(C_0)\ & \emptyset \\
& \Big\downarrow \mathtt{out}(c_A, x) \\
(C_1)\ & \{\mathtt{Rn}(\mathtt{nonce}_A, \mathtt{nonce}_B, \mathtt{nonce}_A)\} \\
& \Big\downarrow \mathtt{in}(c_B, \mathtt{challenge}(x_{\mathtt{nonce}A})) \\
(C_2)\ & \{\mathtt{Rn}(\mathtt{nonce}_A, \mathtt{nonce}_B, \mathtt{nonce}_A)\} \\
& \Big\downarrow \mathtt{out}(c_B, y) \\
(C_3)\ & \{\mathtt{Rn}(\mathtt{nonce}_A, \mathtt{nonce}_B, \mathtt{nonce}_A), \mathtt{Rn}(\mathtt{nonce}_B, \mathtt{nonce}_A, \mathtt{nonce}_B)\} \\
& \Big\downarrow \mathtt{in}(c_A, \mathtt{response}(y_{\mathtt{nonce}B})) \\
(C_4)\ & \{\mathtt{Rn}(\mathtt{nonce}_A, \mathtt{nonce}_B, \mathtt{nonce}_A), \mathtt{Rn}(\mathtt{nonce}_B, \mathtt{nonce}_A, \mathtt{nonce}_B)\} \\
& \Big\downarrow \mathtt{out}(c_A, z) \\
(C_5)\ & \{\mathtt{Rn}(\mathtt{nonce}_A, \mathtt{nonce}_B, \mathtt{nonce}_A), \mathtt{Rn}(\mathtt{nonce}_B, \mathtt{nonce}_A, \mathtt{nonce}_B)\} \\
& \Big\downarrow \mathtt{in}(c_B, \mathtt{data}(z_{\mathtt{key}}, z_{\mathtt{pl}})) \\
(C_6)\ & \{\mathtt{Rn}(\mathtt{nonce}_A, \mathtt{nonce}_B, \mathtt{nonce}_A), \mathtt{Rn}(\mathtt{nonce}_B, \mathtt{nonce}_A, \mathtt{nonce}_B), \mathtt{Ag}(\mathtt{nonce}_B, \mathtt{nonce}_A, \mathtt{key})\}
\end{aligned}
$$

Figure 20: Example trace of claims

$$
\begin{aligned}
\forall_{\mathtt{nonce}_A, \mathtt{nonce}_B, \mathtt{key}, i}.\mathtt{Ag}(\mathtt{nonce}_B, \mathtt{nonce}_A, \mathtt{key}) \mathbin{@} i \implies \\
\exists_{j,k}.\mathtt{Rn}(\mathtt{nonce}_A, \mathtt{nonce}_B, \mathtt{nonce}_A) \mathbin{@} j \\
\wedge\, \mathtt{Rn}(\mathtt{nonce}_B, \mathtt{nonce}_A, \mathtt{nonce}_B) \mathbin{@} k \\
\wedge\, j < i \ \wedge\ k < i
\end{aligned}
$$

Figure 21: Example expression over claims

nonce with an agent: For example, Alice is identified through sending $\mathtt{nonce}_A$, and Bob by $\mathtt{nonce}_B$. This means that the claim $\mathtt{Ag}(\mathtt{nonce}_B, \mathtt{nonce}_A, \mathtt{key})$ expresses that the "sender" of $\mathtt{nonce}_B$ believes that they and the sender of $\mathtt{nonce}_A$ have agreed on the derived key. Since it is protocol dependent when a claim is made, this claim could be made upon the final step of our happy-flow: where Bob receives an encrypted data packet from Alice and decrypts it, therefore establishing that the symmetric key used is `key`. Continuing, a claim similar to agreement is the **running**-claim: $\mathtt{Rn}(m_1, m_2, m_d)$. However, here the claimer only knows that they have sent data themselves m_d to the agent of m_2, there are no signals that is has arrived correctly. For example, $\mathtt{Rn}(\mathtt{nonce}_A, \mathtt{nonce}_B, \mathtt{nonce}_A)$ can be claimed by Alice when they send their own nonce to Bob. These two types of claim can be placed in a trace of claims C for the DY-execution of Figure 18. We use the running claim when sending nonces, and the agreement claim for receiving the first data packet. The resulting C is shown in Figure 20. Finally, a straightforward logical expression over these positioned claims is a claim that, in order to agree over a key, both agents have to contribute a random number. This expression is shown in Figure 21, which expresses that when an agreement claim is made over a key, then there must have been running claims over the nonces somewhere in the past of the trace.

4.4 Mutations

The Puffin tool starts fuzzing from a set of seed traces: a number of Dolev-Yao traces that denote happy-flows in the SUT. These seeds are executed to form an initial corpus. Then the tool mutates traces in the corpus. A proposed mutation is only performed if the new trace passes a simple type check. Every entry in the signature is given a type: for atomics this is simply the type the instanced variable would have, for example an array of bytes for a client random, or a X.509 certificate for the peer certificate. For the other operators the type of the result of the function application is used. This could be an extension for the application for the key share extension, for example. In particular TLS packets are defined as "Messages", which will prevent a package inside a package. This system prevents Puffin from building messages that are highly malformed, and thus very likely to be dropped immediately. Conversely, it could very well be these malformed packages that trigger strange behaviour in the SUT. We expand upon this balance in Section 7.5. In the following, we refer to Figure 22 for examples of every type of mutation Puffin performs, in this figure we use the example protocol and signature given earlier this section. The initial trace is shown in Figure 22a, the possible mutations are shown in red bold font. The possible mutations Puffin can perform are split into two categories: mutations on actions and mutations on terms. An action mutation will either:

- **Repeat** - Insert a copy of an existing action somewhere else in the trace, see Figure 22b.
- **Skip** - Remove an entire action from the trace, see Figure 22c.

Thus, action mutations do not alter the contents of a term. This is in contrast to term mutations, which can:

- **Swap** - Exchanges two (sub-) terms inside the trace. An example of swapping terms is shown in Figure 22d and an example for subterms in Figure 22e. Notably the swapping of whole terms is similar to the outside mutations, however it also enables us to send an existing message to a different agent. Repeat keeps the channel in the copied action as is, while swap does not directly alter it either, swapping the terms in a message to Alice and a message to Bob will have that consequence.
- **Generate** - Replace a term by a randomly generated term, shown in Figure 22f. The term is generated from the operators in the signature Σ of the currently used DY-model. Thus the created term cannot contain variables. This generation is done randomly, albeit bounded by a maximum depth.
- **Replace-Match** - Replace an operator in the term with another operator that has the same arity. This allows Puffin to change atomic data in a term (see Figure 22g) for small but directed changes the concrete packet: Changing a passed version number, for instance, might only change a single byte in a packet, yet it is expected to have a larger impact than bit-flipping a random byte in a packet. At the same time, this mutation can have large effects on generated output: changing an operator that encrypts data with a given key to an operator that concatenates bytes might suddenly send bytes that normally would be consumed in the concretization output and not have been sent normally(the key mentioned earlier).
- **Replace-Reuse** - Replace a term with an existing term present in the current trace, shown in Figure 22h.

- **Remove-and-Lift** - Remove a (sub-) term and replace it with one of its own subterms, see Figure 22i. This mutation is aimed at removing items from a list created through nested operators: Item(x, Item(y, Item(z, Tail))) could return Item(z, Tail), for example.

This list of seven possible mutations allow Puffin to construct a diverse set of inputs from the seed traces. The action mutations allow extending, shortening and reordering of complete traces, which allows us to test the SUTs handling of unexpected messages and repetitions. The trace mutations allow us to test how the SUT reacts to strangely formed messages, for example only changing a single version-byte. Large scale adaptations are also possible, like generating an entire list of extensions into a message. The messages that result from term mutation are not necessarily incorrectly formed, like the version-changing mentioned earlier. However, on the trace level, when we send a message with a different version then the messages before, it could still attempt an invalid execution of the protocol.

These syntactically correct but strange traces allow Puffin to specifically introduce mistakes or mismatches in negotiated of a protocol. This allows for emerging behaviour like an attempted downgrade through negotiation, attempting to use weak cipher suites or sending handshake messages in the wrong order. These attacks (if they would succeed) are related to the protocol logic of the SUT, and are not necessarily memory management issues. Therefore, these DY-trace mutations allow Puffin to explore implementation-level protocol logic faults. A good example of this is the SKIP error Ammann et al. found (see Section 6.1), which let TLS1.3 clients skip verification by omitting a package in the handshake. The fault in the server implementation was unrelated to memory handling, the code simply lacked a conditional statement.

4.5 Dolev-Yao TLS signature

In order to define and mutate traces over TLS, we require a signature that represents messages and data in the TLS protocol. In Puffin 224 terms are defined as part of the TLS signature. The signature contains simple data atomics, like booleans and integers, up to terms representing entire packets of the TLS handshake. One packet is not present in the signature, the TLS 1.3 EndOfEarlyData packet. This means that Puffin is unable to correctly follow the early data specification, as this packet must be sent after sending a ClientHello with the early data extension. We consider this a minor absence, as the present terms are more than enough to build a handshake. Adding this last packet can be considered future work (see Sections 7.4 and 9.5.2).

Puffin includes a number of extensions, including: server name, signature algorithm, extended master secret, session ticket request, key share and renegotiation info extensions. Whilst a number of TLS extensions are not supported, those needed for a correct handshake and for resumption are included. For data fields, Puffin supports three ciphersuites, and thus has these three in its signature. Similarly, three named groups are supported and included. Additionally, Puffin includes fields for three distict certificates, one each for client, server and attacker. Also, for several of these fields empty or wrong instances are included. Whilst we did not include all of the entries in the signature, the entirety allows Puffin to set up a variation of TLS 1.2 and 1.3 handshake routes, with a decent variation in sent data and fields.

Aside from the signature, Puffin also allows some different configurations of the agents themselves during their initial creation for each trace. This notably includes:

- Supported TLS version - Since the TLS 1.3 agents are compatible with TLS 1.2, it is interesting to test the downgrade negotiation logic of the SUT.
- Supported cipher suites and named groups.

$\mathtt{out}(c_A, x_{\mathtt{chall}}) \cdot \mathtt{in}(c_B, \mathtt{challenge}(x_{\mathtt{nonce}A})) \cdot$
$\mathtt{out}(c_B, y_{\mathtt{resp}}) \cdot \mathtt{in}(c_A, \mathtt{response}(y_{\mathtt{nonce}B})) \cdot$
$\mathtt{out}(c_A, z_{\mathtt{data}}) \cdot \mathtt{in}(c_B, \mathtt{data}(\mathtt{enc}(z_{\mathtt{key}}, z_{\mathtt{pl}})))$

(a) Original

$\mathtt{out}(c_A, x_{\mathtt{chall}}) \cdot \mathtt{in}(c_B, \mathtt{challenge}(x_{\mathtt{nonce}A})) \cdot$
$\mathtt{out}(c_B, y_{\mathtt{resp}}) \cdot \mathtt{in}(c_A, \mathtt{response}(y_{\mathtt{nonce}B})) \cdot$
$\mathbf{out}(\mathbf{c_A}, \mathbf{x_{chall}}) \cdot \mathbf{in}(\mathbf{c_B}, \mathbf{challenge}(\mathbf{x_{nonceA}})) \cdot$
$\mathtt{out}(c_A, z_{\mathtt{data}}) \cdot \mathtt{in}(c_B, \mathtt{data}(\mathtt{enc}(z_{\mathtt{key}}, z_{\mathtt{pl}})))$

(b) Repeat

~~$\mathbf{out}(c_A, x_{\mathtt{chall}}) \cdot \mathbf{in}(c_B, \mathbf{challenge}(x_{\mathbf{nonce}A})) \cdot$~~
$\mathtt{out}(c_B, y_{\mathtt{resp}}) \cdot \mathtt{in}(c_A, \mathtt{response}(y_{\mathtt{nonce}B})) \cdot$
$\mathtt{out}(c_A, z_{\mathtt{data}}) \cdot \mathtt{in}(c_B, \mathtt{data}(\mathtt{enc}(z_{\mathtt{key}}, z_{\mathtt{pl}})))$

(c) Skip

$\mathbf{out}(\mathbf{c_A}, \mathbf{x_{chall}}) \cdot \mathbf{in}(\mathbf{c_B}, \mathbf{response}(\mathbf{y_{nonceB}})) \cdot$
$\mathbf{out}(\mathbf{c_B}, \mathbf{y_{resp}}) \cdot \mathbf{in}(\mathbf{c_A}, \mathbf{challenge}(\mathbf{x_{nonceA}})) \cdot$
$\mathtt{out}(c_A, z_{\mathtt{data}}) \cdot \mathtt{in}(c_B, \mathtt{data}(\mathtt{enc}(z_{\mathtt{key}}, z_{\mathtt{pl}})))$

(d) Swap for terms

$\mathtt{out}(c_A, x_{\mathtt{chall}}) \cdot \mathtt{in}(c_B, \mathtt{challenge}(x_{\mathtt{nonce}A})) \cdot$
$\mathtt{out}(c_B, y_{\mathtt{resp}}) \cdot \mathtt{in}(c_A, \mathtt{response}(y_{\mathtt{nonce}B})) \cdot$
$\mathtt{out}(c_A, z_{\mathtt{data}}) \cdot \mathtt{in}(c_B, \mathtt{data}(\mathtt{enc}(\mathbf{z_{pl}}, \mathbf{z_{key}})))$

(e) Swap for subterms

$\mathtt{out}(c_A, x_{\mathtt{chall}}) \cdot \mathtt{in}(c_B, \mathtt{challenge}(x_{\mathtt{nonce}A})) \cdot$
$\mathtt{out}(c_B, y_{\mathtt{resp}}) \cdot \mathtt{in}(c_A, \mathtt{response}(y_{\mathtt{nonce}B})) \cdot$
$\mathtt{out}(c_A, z_{\mathtt{data}}) \cdot$
$\mathtt{in}(c_B, \mathtt{data}(\mathbf{challenge(key)}))$

(f) Generate

$\mathtt{out}(c_A, x_{\mathtt{chall}}) \cdot \mathtt{in}(c_B, \mathtt{challenge}(x_{\mathtt{nonce}A})) \cdot$
$\mathtt{out}(c_B, y_{\mathtt{resp}}) \cdot \mathtt{in}(c_A, \mathtt{response}(\mathbf{x_{nonceA}})) \cdot$
$\mathtt{out}(c_A, z_{\mathtt{data}}) \cdot \mathtt{in}(c_B, \mathtt{data}(\mathtt{enc}(z_{\mathtt{key}}, z_{\mathtt{pl}})))$

(g) Replace-Match

$\mathtt{out}(c_A, x_{\mathtt{chall}}) \cdot \mathtt{in}(c_B, \mathtt{challenge}(x_{\mathtt{nonce}A})) \cdot$
$\mathtt{out}(c_B, y_{\mathtt{resp}}) \cdot$
$\mathtt{in}(c_A, \mathtt{response}(\mathbf{data(enc(z_{key}, z_{pl}))})) \cdot$
$\mathtt{out}(c_A, z_{\mathtt{data}}) \cdot \mathtt{in}(c_B, \mathtt{data}(\mathtt{enc}(z_{\mathtt{key}}, z_{\mathtt{pl}})))$

(h) Replace-Reuse

$\mathtt{out}(c_A, x_{\mathtt{chall}}) \cdot \mathtt{in}(c_B, \mathtt{challenge}(x_{\mathtt{nonce}A})) \cdot$
$\mathtt{out}(c_B, y_{\mathtt{resp}}) \cdot \mathtt{in}(c_A, \mathtt{response}(y_{\mathtt{nonce}B})) \cdot$
$\mathtt{out}(c_A, z_{\mathtt{data}}) \cdot \mathtt{in}(c_B, \mathtt{data}(\mathbf{z_{key}}))$

(i) Remove-Lift

Figure 22: Examples of trace mutations

- Whether they require authentication by either server or client - This is significant for authentication skips, as there is no skip if the agent does not require authentication. Additionally, authentication is often optional for clients, therefore it is useful to test the behaviour of MbedTLS in both cases.
- Certificates - Important for the authentication step.

With the TLS terms, Puffin is able to construct seed traces of various happy-flows in the TLS handshake. Eleven seeds are defined and used as the initial corpus in the fuzzing loop. These include successful handshakes in TLS 1.2 and 1.3. As Puffin can generate TLS messages by itself, the seeds also include happy-flows where Puffin communicates with only a client or only a server. Additionally, a number of variants are included, for example:

- A seed of TLS 1.3 with *ChangeCipherSpec* messages included (these are unneeded, but can be used to mimic a TLS 1.2 session).
- Traces that perform session resumption.
- Traces that require client authentication.

5 Extension to MbedTLS

We extended the Puffin fuzzer to be able to fuzz test MbedTLS 3.6.4. For this, we built a harness instance that initializes and maintains a connection between the Puffin harness interface and instances of MbedTLS. The code of the project is available on Gitlab[6]. We use most of Puffins TLS mapper, as it should be implementation-independent, like the seeds and security oracle. As Puffin was used to successfully fuzz OpenSSL and WolfSSL. However, some modifications had to be made, these are detailed in Section 5.2. In this section, we detail some specific details regarding our implementation. First, we describe Puffins interface in Section 5.1. Then, as said, we explain some necessary modifications to the mapper in Section 5.2. Finally, we describe how Puffin extracts information from MbedTLS in Section 5.3.

5.1 Harness interface

In order to communicate with the harness, Puffin defines an interface in C that the MbedTLS harness must implement. The interface includes the following functions:

- `create` - This function initializes the agent. Various configuration variables are passed through to the harness, which should configure MbedTLS accordingly. These variables are described in Section 4.5. Additionally, this function initializes an input- and an output buffer that Puffin will communicate with through the other harness functions.
- `add_inbound` - This function lets Puffin send data to the SUT. An array of bytes is loaded into the input buffer of the harness, and once the `progress`-function of the harness is called, MbedTLS is fed the contents of said buffer.
- `take_outbound` - This function sends output that is collected from MbedTLS during the `progress` call back to Puffin. In the meantime, the output is stored in a separate buffer in the harness.
- `progress` - The `progress` function is called to "progress" the state of MbedTLS with the inputs Puffin sent earlier through `add_inbound`. Any output that MbedTLS returns is loaded into a buffer in the harness. The harness returns a status report to Puffin, which is directly related to the return values of MbedTLS API-calls. If there are no anomalies, then the harness reports `OK`. If MbedTLS reports that it is busy, or if the end of the input-buffer is reached, the harness reports `IO_WOULD_BLOCK` to indicate that there should be a break in processing. Specifically during the TLS handshake some situations where `IO_WOULD_BLOCK` would be sent `OK` is sent instead, due to MbedTLS frequently requesting five bytes of a packet as a probe for incoming messages and returning. Any other errors are returned to Puffin as `ERROR`, which indicates that the input was rejected.

These functions allow Puffin to send the packets generated from the DY trace to MbedTLS and to receive the returned data.

5.2 Mapper modifications

We found that, although Puffin is not a protocol- or implementation-specific fuzzer, the TLS-mapper had been constructed with certain SUTs in mind, and therefore many variables passed from the mapper would be tailored to the OpenSSL API. In order to simplify some of the pre-processing a MbedTLS-specific harness would have to do, we

[6] `https://gitlab.science.ru.nl/alenaerts/MbedTLSPuffin`

changed the types of some of the initialization variables the Puffin mapper would pass to the harness if MbedTLS was selected as SUT. See also Section 7.7.

5.2.1 Cipher suite variables

The Puffin TLS mapper uses character strings to specify the named groups and cipher suites that the agent would have to use. These variables are specified in OpenSSL-specific naming, which would not be recognized by the MbedTLS API. Therefore we renamed these variables to their IANA identifiers[7]. Additionally, OpenSSL allows users to pass pre-defined and named lists of cipher suites with specific attributes, MbedTLS does not support such functionality. For example, the list "RSA" contains only RSA based ciphers, and the list "NULL" contains suites which have null encryption. When specifying what cipher suites to use in OpenSSL, one can build a list of cipher suites by combining or excluding any of these pre-defined lists: "RSA:!NULL" gives the RSA cipher suites without any null ciphers. MbedTLS does not offer this functionality, therefore we extracted the contents of these pre-defined from an instance of OpenSSL 3.0.2 and defined them as lists in the Puffin mapper-code. Curiously, OpenSSL would always add a certain three "secure" cipher suites at the top of any list, so the list of null ciphers would also contain the three "secure" cipher suites. As a result, when building a set with ":!NULL" in it, the result would not contain the three secure suites. By interpreting the name of the predefined sets, we determined whether to remove (some) of these secure suites from the lists.

5.2.2 Certificate encoding

We found that the bridging of some struct variables between the mapper and the harness would introduce artefact bytes, which would have to be removed by the harness, as the MbedTLS API would not accept such superfluous bytes. This was problematic for the passing of PEM encoded X.509 certificates, it would cause crashes when using the certificates as passed by the mapper directly. Therefore, we had to implement functions that rebuilt the certificate-formatting inside the harness-code.

5.3 Claim extraction & checking properties

Puffin is able to extract certain information from a SUT. Originally, this information was used to create the claims that the objective oracle tests against the TLS (or other protocol of the SUT) security properties. Additionally, since this information often involves the shared secrets and created digests, Puffin is also able to use extracted digests, without having to perform the calculations itself.

The information Puffin extracts are bundled in "Claims" in the form of C structures, which are created by code added manually to the SUT. Whenever the SUT writes a packet we create a claim struct, fill it with all possible information, and extract it to the harness. The harness passes the claim on to the mapper, which uses it as described. Depending on the handshake state pointer of the SUT, we decide what information in the struct is relevant. Some of the information that is extracted:

- **Shared secrets** - Both the TLS 1.2 master secrets and the eleven TLS 1.3 secrets.
- **Handshake state pointer** - This allows Puffin to see if the handshake was successful even if it should not have been.
- **The used cipher suite** - Shows whether some attempt at man-in-the-middle changing the cipher suite was successful.

[7] `https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml`

- **The used signature algorithm** - Like the cipher suite, shows if some attempt at changing was successful.
- **Host and peer certificates** - Shows whether a malformed certificate was accepted as correct.

We have implemented this extraction in the MbedTLS codebase, which creates such a claim struct and sends it to our harness, which passes it to Puffin. Inside the mapper this information is then used by the security oracle to test pre-defined security properties of TLS. These relate to the security values of TLS, given in Section 3.3. The values and there related checks are:

- **Authentication** - If the handshake successfully finishes and authentication is enabled, then one agent must have performed certificate verification of their peers certificate. Puffin checks if the certificate the agent has stored as "peer certificate" is the same as the certificate the peer was given during initialization. If that is not the case, then the verification step was successfully completed with the wrong certificate, and therefore an authentication bypass has occurred. Additionally, an equivalence check is performed on the negotiated signature algorithms.
- **Integrity** - Puffin checks whether the session ids, random values, cipher suites and signature algorithms are equal between client and server.
- **Confidentiality** - Puffin checks whether the shared secrets, cipher suites and random values are equal between client and server.

6 Evaluation

6.1 Model verification

Now that we have the ability to execute Puffin on a TLS signature, we can analyze the possible results of a run. A fuzzer typically does not perform a bounded run, it will keep mutating and testing until the user terminates the program. The tool reports any objectives found along the way, which could point to either a vulnerability in the SUT or an unrelated problem, for example a timeout due to a busy system. However, there is always the chance that the program does not find a single objective after a sufficiently long run (For discussion on how long is sufficient, see Sections 6.1.5 and 7.6). This could be explained in two ways:

1. The SUT does not contain the vulnerabilities we are testing for.
2. The SUT contains vulnerabilities of the type we are testing for, but the fuzzer is not able to find them.

To prevent the latter (2) and verify the fuzzer works correctly, we will execute Puffin on a version of MbedTLS we weaken ourselves. When we add a defect of a certain category into the code and then fuzz, conclusion (1) could never be true. Therefore, if Puffin does not find the inserted defect, we can say that Puffin or our extension is not capable of finding that kind of defect. However, if it is able to find the inserted problem, then it should be able to find similar problems in the code, if they exist. Thus, we build a set of vulnerabilities to verify Puffin's capabilities with. By adding memory management defects or implementation-level protocol logic defects specifically, we can determine if Puffin is capable of finding these vulnerabilities, thus providing the answer for our research questions **RQ1** and **RQ2**. Then we can execute Puffin on a non-mutated instance of MbedTLS. As we know at that point what defects the fuzzer is able to detect, this run will provide the answer for our third research question **RQ3**.

In the next sections, we describe our methodology for this process in steps. First, we describe the different defects we insert into MbedTLS, there are four categories of defects which we explain in their own sections: Memory management defects (Section 6.1.1), implementation-level protocol logic defects (Section 6.1.2), defects that are both (Section 6.1.3), and defects that are neither (Section 6.1.4). Then we explain how we analyze the results of a fuzzing run on the both mutated MbedTLS (Section 6.1.5), and on non-mutated MbedTLS (Section 6.1.6).

6.1.1 Inserted memory management defects

The defects we consider are adapted from CWE-742 [31], a list of possible memory management vulnerabilities in software, adapted from Seacord [29]. From this list we used the base cases. Every defect is positioned in MbedTLS such that the initial seed corpus does not trigger the vulnerability, but a specific aspect of the handshake has to be called to cause the problem. This is done to verify that Puffin is capable actually mutating the traces according to our TLS specification, and is thus capable of reaching different parts in the code of MbedTLS.

The defects we insert are the following:

- **Buffer underflow write** - If a TLS 1.2 server recieves a *ClientHello* with an empty list of cipher suites, we write to the -1th address of a buffer.
- **Buffer underflow read** - If a TLS 1.2 client recieves four malformed *ChangeCipherSpec* messages in a row, we read the -1th address of a buffer.

- **Buffer overflow write** - If a TLS 1.3 server recieves a supported groups encrypted extension with a zero length byte, we write beyond the bound of a buffer.
- **Buffer overflow read** - If a TLS 1.3 server recieves a *ClientCertificate* with the same certificate that the server is currently using, we read beyond the bound of a buffer.
- **Use-after-free** - If a TLS 1.3 server recieves a *ClientHello* where the PSK extension is not the last extension, `free`() is called on a struct that will be used later in the function.
- **Return of stack address** - If a TLS 1.2 server recieves a *NewSessionTicket* that is at least 512 bytes long, then a pointer to a value in stack memory is returned.
- **Free of pointer not at start of buffer** - If a TLS 1.2 client receives a *ServerFinished* message when a ServerHello is expected instead, `free`() is called on a pointer pointing to the middle of a heap allocated buffer.
- **Access of uninitialized pointer** - If a TLS 1.3 server recieves a *ClientHello* without a supported versions extension, a pointer is created and dereferenced without initial assignment.

6.1.2 Inserted implementation-level protocol logic defects

There are two categories of vulnerability we consider here. The first are vulnerabilities that alter data of the internal handshake parameters of MbedTLS in some way. These problems, if not discovered, would most likely not cause a memory management error. But instead disrupt the flow of the handshake in some way and break a guarantee the TLS protocol should provide (see Section 3.3). The defects of this type we add are:

- **Cipher Downgrade** - If a TLS 1.3 server has sent a *ChangeCipherSpec* earlier in the handshake, and now negotiates a cipher suite, then the first cipher suite match found is not used, and searching continues. Protection against downgrading in the negotiated cipher suite is a directly defined security property of TLS. Additionally, using weak encryption also relates to the confidentiality and integrity the protocol provides.
- **Random Tampering** - When a TLS 1.2 client receives more than three malformed *ServerHello* messages, then it stores the packet length byte in the server random. A predictable element in the used nonces weakens the integrity of the communication.
- **Master Tampering** - If a TLS 1.3 client ignores more than 3 extensions received in a CertificateRequest message, then the master secrets first byte is set to zero. An element of predictability in the derived shared secret could harm confidentiality.

The other category of vulnerability are the two authentication skips that Puffin found in WolfSSL. In both of them the verification of the client certificate could be skipped by ommitting parts of the TLS handshake. This is a clear example of a protocol logic defect, in this case clearly the authentication of the client towards the server is not performed correctly. Since this form of authentication is specified in the TLS 1.3 RFC [24], it is an implementation-level protocol logic defect.

- **SIG**- By sending a different signature algorithm in the *Certificate* and following *CertificateVerify* messages the authentication step of the certificate could be skipped in a WolfSSL TLS 1.3 server. This is simply emulated in MbedTLS, since there already is a fallback clause when these two algorithms do not match.

- **SKIP** - A fault in WolfSSL TLS 1.3 where a client, after receiving a certificate request from the server, could forgo sending their certificate in the next message flight, and send a finished message instead. This would then skip the authentication step.

6.1.3 Combination defects

Combination defects require a complex run through the protocol which then reaches some sort of memory management defect. The defects we insert are the problems Ammann et al. [2] found in OpenSSL and WolfSSL which cause memory issues:

- **SDOS1** - A null pointer dereference that occurs in TLS 1.2 servers during renegotiation if the initial *ClientHello* contained a "signature algorithms" extension, but the renegotiation *ClientHello* omits it, whilst including a "signature algorithms cert" extension. Since MbedTLS does not support the "signature algoritms cert" extension, we chose to have the error occur when a "key share" extension was recieved instead. This should not weaken the concept of the attack, occuring if a certain combination of extensions is used through renegotation.
- **CDOS** - A double-free error that could be triggered in WolfSSL clients by receiving a *NewSessionTicket* that is larger than 256 bytes when the client's session cache is not empty. This would cause WolfSSL to enter an error handling block which frees a pointer before reporting failure. Handling this failure would free the same pointer again, causing a crash. MbedTLS does not have a client-side cache, only server-side, therefore, we decided that the most accurate recreation of this error is during session resumption.
- **BUF** - A buffer overflow where a resumption *ClientHello* with at least 12 extensions, but without "supported groups" extension during *Hello Retry* is recieved by a TLS 1.3 server. Additionally list of supported cipher suites sent by the client has to be equal to the internal list of ciphers that the server supports.
- **HEAP** - If a TLS 1.3 server recieves a *ClientHello* message with at least 25 extensions, where at least 12 are "key share" extensions, a buffer over-read occurs.
- **SDOS2** - The last error found by Ammann et al. was a crash if, before session resumption, a non-standard function of the WolfSSL API was used to free the SSL-related memory: `SSL_clear`() was called, rather than the recommended `SSL_free`(). This would cause the server to remove the entire context, rather than the variables inside, without creating a new context structure, and thus would cause a crash once the next client would attempt connection.

6.1.4 Unrelated defects

The final type of defects we insert are problems that are not protocol-logic or memory management related. As such we do not expect that Puffin will be able to find them. We selected these defects from the list of MbedTLS security advisories[8]:

- **Race condition** - Adapted from CVE-2025-52496 [33]. In the "`mbedtls_aesni_has_support`()" function there is a race condition where when two threads call the function at the same time, the second thread could potentially get the wrong result from the function. This would cause the thread to use a weaker implementation of AES or GCM. The race condition was caused by certain

[8] `https://mbed-tls.readthedocs.io/en/latest/security-advisories/`

compiler optimizations which could reorder two variable assignments. We emulate the defect by changing the order of the assignments manually.

- **Side channel** - Adapted from CVE-2025-49087 [32]. During decryption while using the PKCS#7 padding oracle, there is a timing side channel if the original plaintext is larger than the block size. We emulate this defect by reusing an older version of the faulty function.

6.1.5 Mutation result analysis

Once we have inserted a defect, we execute Puffin. The fuzzer will run indefinitely, and therefore one can always argue that, if the objective has not yet been found, Puffin could still find it in the future. During the research, we found that most errors were found in a few thousand queries, and sometimes after about a million. Therefore, we decided that running three million queries would be our limit. Also see Section 7.6 on this point.

Once Puffin has been run and an objective has been found, we verify that the problem Puffin discovered was in fact the vulnerability we added to MbedTLS. The output of Puffin is the objective trace that causes the error. Therefore, we rerun the objective with logging functions enabled. In this white-box environment where we insert code to cause a defect, we can insert a call to logging functions as well. Manual inspection of the logs shows whether the inserted defect was found and whether it triggered the objective oracle. Additionally, if a memory error was found through AddressSanitizer or UndefinedBehaviorSanitizer, or a protocol fault was found through the security oracle, the error messages are used to determine the cause of the problem. Concluding, the analysis of the results is mainly done manually based on error and debug logs. If the objective oracle reports a defect that is not the inserted defect, additional analysis is necessary. If the found defects stems from Puffin related code, we mark the objective as a false positive. Otherwise, we treat the objective as a defect in MbedTLS, which is analyzed as in the following section.

6.1.6 General verification

We proceed to verify a clean instance of MbedTLS. We execute Puffin on an instance of MbedTLS without inserted defects for a specified time. Note that this instance is not a completely fresh release of MbedTLS, since the claim extraction code is inserted into the source code. We empirically determined the bound of our execution of Puffin to be seven million traces (see Section 7.6 and 9.2). If Puffin is able to find an objective, we analyze the resulting trace in the same way we analyzed objective traces for inserted faults, to see if a vulnerability is actually present. Additionally, we execute the objective trace on a completely fresh instance of MbedTLS, to verify that our inserted code was not the cause of the issue. This testing run will provide us with an answer for our final research question **RQ3**.

6.2 Fuzzing results

In this section we show the results of our evaluation. In Section 6.2.1 we provide tables of results of fuzzing runs with inserted defects. In Section 6.2.2 the result of our general verification are presented.

6.2.1 Inserted defect results

We inserted the faults discussed in Section 6.1 into the code of MbedTLS one-by-one and executed Puffin on the MbedTLS library. The Tables (1, 2, 3 and 4) in this section show the results of these executions. Every row in the tablet shows the name of the

fault, which is the name shown in bold in Section 6.1. Then a check mark or cross shows whether we expect Puffin to be able to find the defect under **Expected**. Whether Puffin found an objective during the run is shown under **Found**. If our analysis has shown that the found objective was a false positive, we show a check mark under **FP**. Finally we provide the number of traces that were executed during the run before the objective was discovered, or how many traces were run in total if no objective was found. Table 1 shows the result of inserted memory management faults described in Section 6.1.1. The results of fuzzing implementation-level protocol logic faults from Section 6.1.2 are given in Table 2. Combination faults of Section 6.1.3 are given in Table 3. Finally, Table 4 shows the results of testing faults that are neither memory management or implementation-level protocol logic faults described in 6.1.4.

6.2.2 General verification results

We executed Puffin on an instance of MbedTLS without inserted defects, but with inserted claimer code. After executing only the unmutated seed traces, Puffin reports a coverage of 14.2%. We ran Puffin for a total of 7.004.387 traces in the single run. At the end of the run, the achieved coverage was 30.5%. Three objective traces were found during execution. Each objective was reported as a timeout. Repeating the objective traces did not cause the objective oracles to report an objective again. Therefore, all three objectives are classified as false positives.

Inserted Error	**Expected**	**Found**	**FP**	**Num. Traces**
Underflow Write	✓	✓	×	1.100
Underflow Read	✓	✓	×	426
Overflow Write	✓	✓	×	220
Overflow Read	✓	✓	×	9.101
Use After Free	✓	✓	×	3.517
Return Of Stack Addr	✓	✓	×	44
Free Not At Start	✓	✓	×	18
Use Uninitialized Ptr	✓	×	-	431.229
Use Uninitialized Ptr UBSan	✓	✓	×	52
Null Pointer Dereference	✓	✓	×	122

Table 1: Results of fuzzing memory management defects

Inserted Error	**Expected**	**Found**	**FP**	**Num. Traces**
Cipher Downgrade	✓	×*	✓	41
Random Tampering	✓	✓	×	444
Secret Tampering	✓	×*	✓	85
SIG	✓	×	-	2.903.042
SKIP	✓	×	-	3.026.188

Table 2: Results of fuzzing implementation-level protocol logic defects

Inserted Error	Expected	Found	FP	Num. Traces
SDOS1	✓	×	-	3.159.234
CDOS	✓	✓	×	1.155
BUF	✓	✓	×	10.358
HEAP	✓	✓	×	11.493
SDOS2	✓	✓	×	30

Table 3: Results of fuzzing combination defects

Inserted Error	Expected	Found	FP	Num. Traces
Race Condition	×	×	-	3.278.297
Side Channel	×	×	-	3.200.693

Table 4: Results of fuzzing defects unrelated to memory management and protocol logic

7 Discussion

In this section we analyse the results of our fuzzing runs shown in Section 6. We combine this discussion with an explanation on possibly impactful decisions we have made throughout the project. We start with describing difficulties with Puffins security oracle in Section 7.1. Then we explain the fuzzers reliance on memory error detection tools in Section 7.2. Section 7.3 analyses the coverage results we received from Puffin. Then, in the next sections we discuss several aspects that could influence that coverage: Puffins TLS specification (Section 7.4), limiting factors in the mutations (Section 7.5) and limits in experiment duration (Section 7.6). We finish in Section 7.7 with an analysis of aspects in the harness we have changed, and how we negate this from impacting Puffins results.

7.1 Security oracle weakness

As Table 2 shows, Puffin reported false positive objectives for the cipher downgrade and master tampering defects. This became clear upon rerunning the objective traces with logging functions enabled, as the execution did not pass the injected code. Additionally, Puffin was unable to find the inserted SIG and SKIP defects. Through analysis of the objective traces and inspection of the security oracle, we found that the oracle lacked sufficient checking of claim data. The oracle as of now will make the following mistakes:

- Recognizing different sessions - The oracle will deem a trace which contains two separate sessions in some way as a security violation. This situation can occur because Puffin is capable of setting up TLS sessions with just a single agent through its internal mapper. The second session can be created through either a complete second TLS handshake, or through resumption. The oracle does not check contain a thorough enough check if the claims were created during the same session. The other checks in the oracle will then fail, as they are simple equivalence checks of parameters that will be obviously different between sessions, like the random bytes.

- Cipher selection logic - In order to detect cipher suite downgrades, the security oracle determines on its own which of the cipher suites the agents should have selected. The used cipher suite and the the list of cipher suites which the agents support are extracted through the claim system during a trace (these lists can be different from the list sent by Puffins mapper during agent initialisation). The lists of cipher suites are ordered by agent preference. The oracle checks if the used

cipher suite is the first (and thus highest in preference) suite which both client and server list as supported. However, this is too simple of a method to determine the correct cipher suite, as there is no requirement in the RFC which states that the highest preferred suite should actually be chosen. Instead, in the case of a MbedTLS TLS 1.3 server, it will prefer TLS 1.3 cipher suites over TLS 1.2 cipher suites, even if that is not the first shared cipher suite in the list. This is not indicative of a defect, yet the security oracle will mark it as a security violation.

These issues cause some of our experiments for purely implementation-level protocol logic defects to effectively fail, since we would have to analyze too many false positive objectives. We have attempted to mitigate this problem for the random tampering insertion by disabling the problematic security oracle checks. Puffin was able to correctly find an objective trace that caused a change in the random bytes without any false positives. Therefore, we think that these problems with the security oracle could be mitigated by improving the performed checks or by extracting information from the SUT at other points in the handshake. However, we leave this process for future work (see Section 9.4). At the moment, this means that Puffin is not able to correctly find the SIG, SKIP, cipher suite downgrade and master tampering defects, and thus that its capability of finding implementation-level protocol logic defects without memory mismanagement is reduced.

7.2 Uninitialized pointer use and sanitizer reliance

Table 1 shows that Puffin was not able to find the initial uninitialized pointer defect. This can be explained because Puffin relies on LLVMs AddressSanitizer [9] to detect memory errors, and AddressSanitizer does not detect this particular defect. As such, Puffin would only be able to detect the problem if it directly causes a crash, which was not the case here. To mitigate this issue, we added another detector to our runs: LLVMs UndefinedBehaviorSanitizer [10], which was able to detect the uninitialized pointer and also the null pointer dereference defects. This process highlights a fundamental issue of using Puffin: a reliance on AddressSanitizer, UndefinedBehaviorSanitizer, or some other detector tool to actually find the defects that fuzzing through Puffin can reach. This is further highlighted in one of our experiments where an inserted obvious buffer overflow was not found. After analyzing why, we found that in compilation, the variable where the overflow occurred was not being instrumentalized by AdressSanitizer, which meant that no red zones were created, and thus the overflow was not detected. We had to tweak some compilation flags for the error to be detectable. These issues are expected with both AddressSanitizer, UndefinedBehaviorSanitizer and fuzz testing as a whole, as they do not guarantee completeness. As such, we can not completely solve this issue within Puffin, only minimize the effects through experimentation.

7.3 Coverage results

As the result of our general validation run shows, Puffin achieved a 30.5% coverage of MbedTLS. As said in Section 4.2, this is coverage of edges in the control flow graph of the code of MbedTLS. While this figure is somewhat in line with the code coverage that Ammann et al. reached with Puffin on OpenSSL (16.0%) and WolfSSL (16.9%), it is not relevant to compare Puffins coverage over MbedTLS with other implementations. In our case, we can explain this relatively low figure with the method of coverage tracking. At this moment, we use LLVMs SanitizerCoverage [11], which instrumentalizes the entire

[9] `https://clang.llvm.org/docs/AddressSanitizer.html`
[10] `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`
[11] `https://clang.llvm.org/docs/SanitizerCoverage.html`

MbedTLS code with coverage trackers. This includes functions which Puffin will never reach, like API-functions the harness does not call or cryptographic algorithms which Puffin does not support. As MbedTLS markets itself with a small code base suitable for embedded systems, it makes sense that Puffin reports a higher edge coverage in MbedTLS than OpenSSL, as OpenSSL might not focus as much on a minimal code base. In order to make these comparisons more useful, a different method of coverage would have to be used. This is left as future work (see Section 9.3), as well as a comparison with different fuzzers on the same code. Regardless of other fuzzers and implementations, the fact that throughout our general fuzzing run coverages increases from 14.2% to 30.5% means that the mutations of Puffin manage to effectively create inputs that reach other cases in the code of MbedTLS.

7.4 Protocol domain

The specification for TLS Puffin uses contains operators aimed at testing the handshake, key update and resumption functionality specifically. We choose to target the handshake because it is the location where the parameters of the TLS session are sent and negotiated. As such, this is the area where we expect changing values or shuffling packets will be the most impactful. Aside from the aspects we focus on there are three areas of possible interest in TLS:

- Application data traffic: our signature contains only one operator that represents a message with application data. While our seeds each contain a term with a successful application data message, this setup does not specifically motivate extensive sending of data. The data that *is* sent is decrypted by Puffin. If decryption is successful we know that encryption was successful as well. Therefore, we can be sure that the transmitting of data is done correctly. The contents of the payload are not processed in another way, and therefore are not relevant for a correct functioning of TLS.
- Key update timing: TLS 1.2 and 1.3 both have methods to refresh the shared ephemeral secret keys. In TLS 1.2, this is through renegotiation after a *HelloRequest* message, which should restart the handshake. In TLS 1.3 this is through a *KeyUpdate* message, after which both hosts should derive the next generation of keys from the parameters they have already negotiated. Both these messages are contained in our specification, and thus are tested. However, the TLS 1.3 RFC also specifies that the *KeyUpdate* message should be sent prior to processing a certain amount of data according to Luykx et al. [18]. The bounds Luykx et al. give are however, in the order of terabytes. As such, implementing a check for the correct rekeying behaviour in Puffin would grow fuzzing time to numbers unfeasible for our current setup. Therefore we decided not to test the rekeying limits for MbedTLS.
- Post-handshake authentication: In TLS 1.3 a client can offer a "post_handshake_auth" extension in the initial message flight to the server. The server can then send a *CertificateRequest* message after the handshake has been completed. The client has to respond with the normal trio of certificate messages: *Certificate*, *CertificateVerify* and *Finished*. The server uses this information to perform the normal authentication. As this scheme is functionally identical to the normal authentication, except later in the session, we assume that the same logic is used internally to process the certificate messages, and therefore we do not lose meaningful confidence in our results by implementing the "post_handshake_auth" extension.

Concluding, we find that our focus on the handshake does not significantly harm the validity of our experiment or skip important aspects of the TLS protocol.

7.5 Type checking and mutation domain

When mutating, Puffin performs a type check to verify that a term does not contain nonsense. This is somewhat counter-intuitive to the goal of fuzzing, as fuzzing attempts to increase coverage through randomly mutating inputs, and it could very well be these nonsense inputs that reach unexpected code. This touches upon an issue native to the mutation in fuzzing: the balance of randomness versus efficiency. If a SUT immediately drops inputs that are slightly malformed, it is not efficient to keep producing traces which we know are barely processed. Conversely, if we limit mutating a field with, for example, a number between one and ten, we will never find out what happens if we fill in a alphabetical letter instead.

Puffin is a gray-box fuxxer which also requires knowledge of the protocol of the SUT. As such, we consider it more in line with the capabilities of Puffin to create packages that syntactically conform with the TLS protocols. However, in the process of inserting memory errors, we found that often times the code block where we inserted the error was unreachable due to restrictiveness in mutating. For example: one attempted error insertion was an erroneous `free`()-call if the received server random contained only zeroes. We realized after an unsuccessful test run that Puffin was configured to only send correctly generated random values as random, and would effectively never send just zeroes. While the result of this does not harm our answer to **RQ1**, since we show the capability to find a double `free`()-call elsewhere, it does show that the domain Puffin tests in TLS could be improved. Increasing the coverages for edge-cases of TLS is a possibility for future work, see Section 9.5.2.

7.6 Experiment limits

In theory a fuzzing test run lasts forever. Puffin (and some LibAFL configurations) only stores traces that add coverage or that achieve an objective. A trace that does not achieve either is dropped. This means that there is no measure to determine whether an uninteresting trace has been run before. The reason behind dropping uninteresting traces is due to efficiency. Fuzzing is a method of brute-forcing inputs for a SUT. With that in mind, a lookup through every previously run trace would slow down executing of new traces. There are, in the case of Puffin, seven different mutations, nearly two hundred terms, and a trace is mutated up to fifteen times before running. Storing every trace or a hash would grow into hard to process sizes.

As a fuzzing run does not end naturally, if no objective is found, some termination bound has to be chosen. In this, we chose to be practical. Our process of mutation testing requires us to perform multiple testing runs, As such, we can not spend days fuzzing a single defect. We decided a simple time bound: if an error was not found in three hours of fuzzing on our machine, we consider the error not detectable. The coincides with about three million traces run, which has shown to be long enough for most of the errors we expected to be detectable. However, the SDOS1, SIG and SKIP defects were not found. The SDOS1 defect is noted by Ammann et al. [2] as the most complex to reach defect they have discovered, requiring tens of millions of traces ran to find. We suspect that with a higher termination bound, these defects could still be detected. See also future work Section 9.2

7.7 Harness variable types

During the implementation process, we changed the type of some variables in Puffins harness interface, which Puffin uses to send initialisation information for the SUT from the mapper to the harness. Naturally, we need to be sure that this change does not

introduce unwanted behaviour into the fuzzer's interaction with the SUT. We changed the variable types at two locations:

- The list of cipher suites, which were in an OpenSSL specific format [12]. They were sent as either:
  - Identifier strings for single cipher suites, who we mapped directly to their IANA identifiers [13], which are integers.
  - A set of cipher suites which are referred to by a single name. For example, "DHE" would return all cipher suites that use ephemeral Diffie-Hellman key agreement. We altered the mapper to return a set of IANA identifiers for each of these named sets. Additionally, OpenSSL, and thus Puffin, allows users to construct a customized set of cipher suites by concatenating or negating these named sets, so "DHE:!NULL" would return the cipher suites using ephemeral Diffie-Hellman, excluding any suites that contain a NULL cipher. We use set unions and intersections to make sure the correct set of suites is built if such a customized string is used.

  To make sure that no different cipher suite could be selected from the handshake negotation then Puffin expects, we have done the following:

  - We tested the mapping for every individual cipher suite currently supported. As there are only 5 supported suites, this was rather easily done and verified correct.
  - Puffin uses OpenSSL's sets of cipher suites during initialization. However, this is through a hardcoded character string, which is not altered in any way. We also verified that this string maps to intended suites and in the correct order manually.
  - No other sets of cipher suites are used as of writing. For any future extensions, a simple testing script could be written to ensure that single suites all map correctly to the expected integers.

- The X.509 certificates, we rebuilt these inside the harness from a string sent by the mapper. This process could cause differences in the expected and observed behaviour of the SUT. For example, if a correct certificate is sent by the mapper, and our rebuilding is performed incorrectly, the SUT will refuse it, whilst Puffin does not expect a refusal. We verify the correctness of our rebuiler function by simply testing it with the certificates and keys Puffin can use. Since there are only three sets of one certificate and one key each, this could be done by changing the seed traces and checking the debug logs of MbedTLS manually. Every certificate was correctly parsed by MbedTLS. Additionally, our function is only used on initialization of a SUT. Should a certificate built through our function get refused, it will be reported by Puffin as an error in the fuzzer. Only errors that occur after the trace starts are classified as errors in the SUT. We have not encountered an occurrence of this happening after tens of millions of trace executions, and thus initializations. Since there are only six hardcoded files to parse, which we have also verified manually to work and be included in mutations, we consider this evidence that our certificate building function works correctly.

We conclude that our altering of the Puffin harness interface does not harm the validity of our experiments, both changes we have manually validated as correct.

[12] `https://docs.openssl.org/master/man1/openssl-ciphers/`
[13] `https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml`

8 Conclusion

In this section we analyze the results of our validation runs of Section 6 and the observations made regarding other aspects of the research in Section 7 to answer our research questions. We answer each subquestion in the following subsections, and discuss relevant observations from the results in the associated section of their research question. For completeness, our research questions are:

RQ1. What memory management faults is Puffin able to detect in MbedTLS? (Section 8.1)

RQ2. What implementation-level protocol logic faults is Puffin able to detect in MbedTLS? (Section 8.2)

RQ3. Which existing memory management faults and implementation-level protocol logic faults does Puffin find in MbedTLS, if any? (Section 8.3)

We finish the chapter with a general conclusion of the research and the work performed (Section 8.4).

8.1 Finding memory management defects

To answer question **RQ1** we first look at the results of fuzzing inserted memory management defects. Puffin was able to find every defect in less than ten thousand queries run. The one exception to this was the uninitialized pointer defect without UndefinedBehaviorSanitizer. See Section 7.2 for more details on this issue. Regardless, the other memory management issues are discovered quickly by Puffin, each with the associated sanitizer messages and error traces, which allowed us to verify that the errors detected were caused by the inserted defects. For the combination defects, the memory management aspect of these problems are simply other instances of the types of faults we inserted as memory management faults. As such, these were detected through the sanitizers. Concluding, we find that Puffin is capable of detecting memory management faults in MbedTLS. These have to be covered by the TLS specification given in DY-terms. Additionally, the defect also has to be detectable by the used sanitizers or cause a crash directly. According to our experiments, Puffin could find buffer over- and underflows, use-after-free, uninitialized pointer use, return of stack pointer, free not at start of buffer and null pointer dereferences.

8.2 Finding implementation-level protocol logic faults

For question **RQ2** we analyze the results of fuzzing the inserted implementation-level protocol logic defects. Puffin was not able to detect the inserted SIG and SKIP errors in MbedTLS within three million traces. Additionally, Puffin had difficulty detecting the the cipher downgrade and secret tampering defects. This is due to the security oracle reporting false positives. However, when we limit MbedTLS to only TLS 1.2, the selection of cipher suites does not report a false positive, and the default could be detected correctly. We do not consider the functioning of the security oracle a fundamental problem with Puffins methodology and think that with additional logic, detection could be vastly improved. We consider this an option for future work, see Section 9.4. By disabling checks in the security oracle that generate false positives, Puffin is able to find the random tampering defect.

Puffin was able to correctly find most combination defects, with the exception of SDOS1. This shows that Puffin is capable of finding complex paths through the TLS handshake. It is still unsure whether Puffin is able to reach the SDOS1, SIG and SKIP

defects in MbedTLS, as we have only run five million traces with SDOS1 inserted, and three million traces for SIG and SKIP each. We suspect that the SDOS1, SIG and SKIP defects could be detected in MbedTLS with a larger threshold of executed traces. However, due to time constraints, we leave this for future work (see Section 9.2). Concluding, we find that Puffin is capable of detecting implementation-level protocol logic faults, if they are covered by the TLS specification given in Puffins DY terms. These defects also have to be detectable by the used sanitizers if they are also memory management faults, and otherwise Puffins security oracle can find defects in restrictive configurations.

8.3 Fuzzing MbedTLS

From conclusions for **RQ1** and **RQ2** follows that Puffin is able to find memory management faults and implementation-level protocol logic faults, on the following conditions:

- A memory fault either causes a crash or is recognized by AddressSanitizer and UndefinedBehaviorSanitizer.
- The purely implementation-level protocol logic fault that Puffin can detect is tampering with the random bytes, since we disabled logic that returns false positives in the security oracle.
- Puffin is only able to detect faults in parts of the TLS protocol that are covered by the internal TLS specification.

We executed Puffin on a non-mutated version of MbedTLS for seven million traces, where no defects were found. We answer **RQ3** with the conclusion as follows: While Puffin should have been able to detect a variety of memory management faults, combination faults, and a single purely implementation-level logic fault, it did not find any existing faults.

8.4 General conclusion

In this work we have tested the capabilities of the Puffin fuzzer to find memory management faults and implementation-level protocol logic faults in MbedTLS. We conclude our research summarizing the answer to our research question: Puffin is able to detect a variety of both memory management and implementation-level protocol logic faults in MbedTLS by mutating traces in the provided domain of TLS as DY terms. However, no such faults were found in the production code of MbedTLS. As such, we conclude that MbedTLS does not contain these types of defects. This means that MbedTLS, to the extent we have tested it with Puffin, does not contain any defects that harm the integrity, confidentiality and authentication guarantees of TLS. This conclusion could be extended by changing the security oracle of Puffin to significantly improve the detection of different types of implementation-level protocol logic defects. However, due to time constraints, this has been added as future work in Section 9.4.

9 Future Work

Our experimental setup has potential to be expanded or iterated upon. This could lead to more accurate reported results, more coverage over the TLS protocol, more types of faults to be detected, or different areas where MbedTLS is used to be explored. In this section we describe some options for future work we have identified through our project. We start with extending Puffin to another protocol in Section 9.1. Then we detail some areas where our research domain could be expanded: Length of fuzzing in Section 9.2, coverage calculation in Section 9.3 and expanding the security oracle in Section 9.4. Finally, we suggest what could be changed so Puffin reaches a greater subset of the MbedTLS code in Section 9.5.

9.1 Extending Puffin to OpenVPN

This project initially started as a project to fuzz test OpenVPN. OpenVPN has a large code base with several legacy features. Additionally, the protocol specification of OpenVPN is unclear, as it is poorly documented. Ammann et al. [2] note that Puffin could be used independently of protocols, but themselves have only finished building and testing the TLS extension of Puffin (a SSH extension is a work in progress as of writing). It would be interesting to research whether the OpenVPN protocol could be specified in DY-terms, and whether the tool could be tested through Puffin.

9.2 Fuzzing length

A simple option for future work is to repeat our experiment for SDOS1, SIG, SKIP and our general run with a higher maximum bound for traces. This could be done by running Puffin on a more powerful computer or compute cluster. Ammann et al. [2] for example ran Puffin on a server with 500 GB of RAM and 32 CPU cores, which significantly improves the number of traces run over time. Ammann notes that the number of traces run to find SDOS1 would number in the order of seven digits. Running Puffin for more traces on a clean instance of MbedTLS could achieve more coverage and potentially discover more defects.

9.3 Coverage methods

Puffin tracks coverage through the SanitizerCoverage of LLVM. At the moment, the entire codebase of MbedTLS is instrumentalized through this tool. This causes a significant amount of "unreachable" code for our fuzzer to be annotated with coverage instructions. For example, Puffin currently supports three cipher suites. However, MbedTLS supports many more, these all are annotated by SanitizerCoverage. The result is that the reporting of coverage is not as accurate as it could be. SanitizerCoverage supports a whitelisting and blacklisting of code functions, which would allow a more controlled report of coverage. It would be interesting to research how much of the "protocol logic" Puffin actually reaches.

9.4 Security oracle strengthening

Our fuzzing of purely implementation-level protocol logic faults found a number of false positives, as logic of the security oracle is too simple. We do not consider the problems we encountered with the current iteration of the oracle to be some fundamental problem with the methodology. Instead we suspect that adding more extensive checking of conditions and extracting claims from the SUT at other points in the TLS handshake will allow Puffin to correctly process potential security violations. Therefore, it would

be interesting to rerun our experiments with MbedTLS after strengthening the security oracle. Thus checking once again if MbedTLS contains no implementation-level protocol logic defects.

Additionally, extending the security oracle to include the security specifications as given in the Dolev-Yao model would greatly improve the potential checking of security violations. Currently the security oracle performs some equivalence checks on extracted values. The Dolev-Yao model, with its origin in symbolic verification, could be used to prove more complex properties over TLS instead. The work of Ammann et al. [2] suggests so-called agreement and running claims as used in TAMARIN [19].

Finally, there is a potential for the security oracle to find specification-level protocol logic faults in a SUT. For example, for a protocol that guarantees confidentiality for data payloads, if there is some legal path through the state machine of that sends a payload as plaintext, then specifying in the security oracle that all payloads must be encrypted should find this problem. We consider that other tools are better suited to finding specification-level faults, like symbolic verification. Nevertheless, it could be interesting to research if there are situations where this potential is useful.

9.5 MbedTLS completeness, configurations and versions

Our coverage of MbedTLS could be increased in a number of ways. The next sections cover some of our suggestions.

9.5.1 MbedTLS major versions

Our experiments were conducted on version 3.6.4 of MbedTLS with the default configuration. At the start of our project 3.6.4 was the most recent release of MbedTLS. However, during the course of our project, MbedTLS 4.0.0 was released, which introduces a "major codebase restructuring" [14]. It would be interesting to test other versions of MbedTLS.

9.5.2 TLS signature completeness

As noted in Section 4.5 and discussed in Section 7.4, the current TLS signature used in Puffin does not include the EndOfEarlyData message, and therefore lacks the capability to send every single TLS 1.2 and 1.3 handshake message. Additionally, Puffin includes three cipher suites and three named groups in its signature. Finally, not every TLS extension is usable by Puffin. In order to improve Puffins coverage of TLS, potential future work could be to expand the mapper with the ability to process these messages and cipher suites. However, a downside to adding more signature entries is that Puffin will be able to perform even more mutations and thus the potential state space will grow. It might then take longer to perform the precise mutations that reach a certain defect. However, this problem can be somewhat mitigated by selecting what packets, variables and extensions to include in the signature for a specific run.

9.5.3 MbedTLS Configurations

As mentioned in Section 9.1, this work originally started as a project to test OpenVPN. Another way this aspect could return in future work is by fuzzing MbedTLS as it is being used in OpenVPN. Potentially other software that uses MbedTLS could be examined to see how MbedTLS is configured for these projects and which functions of the MbedTLS library they use. It would then be interesting to research whether Puffin is able to find defects in those instances of MbedTLS. If those specific instances contain

[14] `https://github.com/Mbed-TLS/mbedtls/releases/tag/mbedtls-4.0.0`

some vulnerability, then the software the MbedTLS configuration is used in could be vulnerable as well.

References

[1] Alnahawi, Nouri, Johannes Müller, Jan Oupický, and Alexander Wiesmaier: *A comprehensive survey on post-quantum TLS*. IACR Communications in Cryptology, July 2024. https://inria.hal.science/hal-04845617.

[2] Ammann, Max, Lucca Hirschi, and Steve Kremer: *DY fuzzing: Formal Dolev-Yao models meet cryptographic protocol fuzz testing*. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1481–1499, 2024.

[3] Atlidakis, Vaggelis, Patrice Godefroid, and Marina Polishchuk: *Restler: Stateful rest api fuzzing*. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758, 2019.

[4] Bastani, Osbert, Rahul Sharma, Alex Aiken, and Percy Liang: *Synthesizing program input grammars*. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 95–110, New York, NY, USA, 2017. Association for Computing Machinery, ISBN 9781450349888. https://doi.org/10.1145/3062341.3062349.

[5] Bhargavan, Karthikeyan, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters: *Formal security and functional verification of cryptographic protocol implementations in Rust*. Cryptology ePrint Archive, Paper 2025/980, 2025. https://eprint.iacr.org/2025/980.

[6] Blanchet, Bruno: *Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif*, pages 54–87. Springer International Publishing, Cham, 2014, ISBN 978-3-319-10082-1. https://doi.org/10.1007/978-3-319-10082-1_3.

[7] Bruyère, Véronique, Bharat Garhewal, Guillermo A. Pérez, Gaëtan Staquet, and Frits W. Vaandrager: *Active learning of mealy machines with timers*. In Prabhakar, Pavithra and Andrea Vandin (editors): *Quantitative Evaluation of Systems and Formal Modeling and Analysis of Timed Systems - Second International Joint Conference, QEST+FORMATS 2025, Aarhus, Denmark, August 26-28, 2025, Proceedings*, volume 16143 of *Lecture Notes in Computer Science*, pages 42–61. Springer, 2025. https://doi.org/10.1007/978-3-032-05792-1_3.

[8] Cremers, Cas, Marko Horvat, Sam Scott, and Thyla van der Merwe: *Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication*. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 470–485, 2016.

[9] Daniel, Lesly Ann, Erik Poll, and Joeri de Ruiter: *Inferring OpenVPN state machines using protocol state fuzzing*. In *2018 IEEE European Symposium on Security and Privacy Workshops*, pages 11–19, 2018.

[10] Daniele, Cristian, Seyed Behnam Andarzian, and Erik Poll: *Fuzzers for stateful systems: Survey and research directions*. ACM Comput. Surv., 56(9), April 2024, ISSN 0360-0300. https://doi.org/10.1145/3648468.

[11] Dolev, D. and A. Yao: *On the security of public key protocols*. IEEE Transactions on Information Theory, 29(2):198–208, 1983.

[12] Draeger, Joachim: *Sigma-algebra (Computer Science)*. In *Encyclopedia of Mathematics*. EMS Press, April 2013. http://www.encyclopediaofmath.org/index.

php?title=Sigma-algebra_(Computer_Science)&oldid=29689, Archived at Wayback Machine https://web.archive.org/web/20250423221312/https://encyclopediaofmath.org/index.php?title=Sigma-algebra_(Computer_Science)&oldid=29689, capture dated 23 April 2025, accessed on 20 March 2026.

[13] Durumeric, Zakir, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman: *The matter of heartbleed*. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery, ISBN 9781450332132. https://doi.org/10.1145/2663716.2663755.

[14] Fioraldi, Andrea, Dominik Maier, Heiko Eißfeldt, and Marc Heuse: *AFL++ : Combining incremental steps of fuzzing research*. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. https://www.usenix.org/conference/woot20/presentation/fioraldi.

[15] Hu, Fan, Jiangan Ji, Hui Shu, Zheming Li, Tieming Liu, and Chao Zhang: *Formatted stateful greybox fuzzing of tls server*. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 151–160. IEEE, 2024.

[16] Kitagawa, Takahisa, Miyuki Hanaoka, and Kenji Kono: *AspFuzz: A state-aware protocol fuzzer based on application-layer protocols*. In *The IEEE symposium on Computers and Communications*, pages 202–208, 2010.

[17] Liu, Chiang, Hongpei Zheng, Xin Zhang, Dapeng Ju, Dongsheng Wang, Yinqian Zhang, and Trevor E. Carlson: *SSBleed: Non-speculative side-channel attacks via speculative store bypass on Armv9 CPUs*. 2026 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2026.

[18] Luykx, Atul and Kenneth G. Paterson: *Limits on authenticated encryption use in TLS*. Cryptology ePrint Archive, Paper 2024/051, 2024. https://eprint.iacr.org/2024/051.

[19] Meier, Simon, Benedikt Schmidt, Cas Cremers, and David Basin: *The TAMARIN prover for the symbolic analysis of security protocols*. In Sharygina, Natasha and Helmut Veith (editors): *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg, ISBN 978-3-642-39799-8.

[20] Miller, Barton P., Lars Fredriksen, and Bryan So: *An empirical study of the reliability of UNIX utilities*. Commun. ACM, 33(12):32–44, December 1990, ISSN 0001-0782. https://doi.org/10.1145/96267.96279.

[21] Möller, Bodo, Thai Duong, and Krzysztof Kotowicz: *This poodle bites: Exploiting the ssl 3.0 fallback*. 2014.

[22] Pan, Yan, Wei Lin, Yubo He, and Yuefei Zhu: *Coverage-guided differential testing of TLS implementations based on syntax mutation*. PLOS ONE, 17(1):1–16, January 2022. https://doi.org/10.1371/journal.pone.0262176.

[23] Petsios, Theofilos, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana: *NEZHA: Efficient domain-independent differential testing*. In *2017 IEEE Symposium on security and privacy (SP)*, pages 615–632. IEEE, 2017.

[24] Rescorla, Eric: *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446, August 2018. https://www.rfc-editor.org/info/rfc8446.

[25] Rescorla, Eric and Tim Dierks: *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246, August 2008. https://www.rfc-editor.org/info/rfc5246.

[26] Ruiter, Joeri de and Erik Poll: *Protocol state fuzzing of TLS implementations*. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association, ISBN 978-1-939133-11-3. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[27] Sardar, Muhammad Usama, Arto Niemi, Hannes Tschofenig, and Thomas Fossati: *Towards validation of TLS 1.3 formal model and vulnerabilities in Intel's RA-TLS protocol*. IEEE Access, 12:173670–173685, 2024.

[28] Schumilo, Sergej, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz: *Nyx-net: network fuzzing with incremental snapshots*. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 166–180, New York, NY, USA, 2022. Association for Computing Machinery, ISBN 9781450391627. https://doi.org/10.1145/3492321.3519591.

[29] Seacord, Robert C: *The CERT C Secure Coding Standard*. Addison-Wesley Educational Publishers Inc., 2008.

[30] Swierzy, Ben, Felix Boes, Timo Pohl, Christian Bungartz, and Michael Meier: *SoK: Automated software testing for TLS libraries*. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ARES '24, New York, NY, USA, 2024. Association for Computing Machinery, ISBN 9798400717185. https://doi.org/10.1145/3664476.3670871.

[31] The MITRE Corporation: *CWE category: CERT C secure coding standard (2008) chapter 9 - memory management (MEM)*, 2008. https://cwe.mitre.org/data/definitions/742.html, Accessed: 2026-02-25.

[32] The MITRE Corporation: *CVE-2025-49087*, 2025. https://www.cve.org/CVERecord?id=CVE-2025-49087, Accessed: 2026-02-25.

[33] The MITRE Corporation: *CVE-2025-52496*, 2025. https://www.cve.org/CVERecord?id=CVE-2025-52496, Accessed: 2026-02-25.

[34] Vanhoef, Mathy and Frank Piessens: *Key reinstallation attacks: Forcing nonce reuse in WPA2*. In Thuraisingham, Bhavani, David Evans, Tal Malkin, and Dongyan Xu (editors): *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1313–1328. ACM, 2017. https://doi.org/10.1145/3133956.3134027.

[35] Walz, Andreas and Axel Sikora: *Exploiting dissent: Towards fuzzing-based differential black-box testing of TLS implementations*. IEEE Transactions on Dependable and Secure Computing, 17(2):278–291, 2020.

[36] Ye, Katherine Q., Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel: *Verified correctness and security of mbedtls HMAC-DRBG*. CoRR, abs/1708.08542, 2017. http://arxiv.org/abs/1708.08542.