DY Fuzzing: Formal Dolev-Yao Models Meet Protocol Fuzz Testing

Max Ammann*
Independent Researcher &
Trail of Bits
max@maxammann.org

Lucca Hirschi
Inria Nancy Grand-Est
Université de Lorraine, LORIA, France
lucca.hirschi@inria.fr

Steve Kremer
Inria Nancy Grand-Est
Université de Lorraine, LORIA, France
steve.kremer@inria.fr

v1.0† — January 18, 2023

Abstract— **Critical and widely used cryptographic protocols have repeatedly been found to contain flaws in their design and their implementation. A prominent class of such vulnerabilities is logical attacks, *i.e.* attacks that solely exploit flawed protocol logic. Automated formal verification methods, based on the Dolev-Yao (DY) attacker, excel at finding such flaws, but operate only on abstract specification models. Fully automated verification of existing protocol implementations is today still out of reach. This leaves open whether widely used protocol implementations are secure. Unfortunately, this blind spot hides numerous attacks, notably recent logical attacks on widely used TLS implementations introduced by implementation bugs.**

We answer by proposing a novel and effective technique that we call DY model-guided fuzzing, which precludes logical attacks against protocol implementations. The main idea is to consider as possible test cases the set of abstract DY executions of the DY attacker, and use a mutation-based fuzzer to explore this set. The DY fuzzer concretizes each abstract execution to test it on the program under test. This approach enables reasoning at a more structural and security-related level of messages (*e.g.* decrypt a message and re-encrypt it with a different key) as opposed to random bit-level modifications that are much less likely to produce relevant logical adversarial behaviors. We implement a full-fledged and modular DY protocol fuzzer. We demonstrate its effectiveness by fuzzing three popular TLS implementations, resulting in the discovery of four novel vulnerabilities.

1. Introduction

Cryptographic protocols are very hard to get right. Critical and widely used protocols have been repeatedly found to be flawed in their design and their implementation, usually with dramatic consequences. For decades, this has concerned security researchers, who have identified different relevant classes of attacks and methods to prevent them.

Logical attacks and formal verification. Since the 1980s [30], the formal methods community has identified and mathematically defined the class of ***logical attacks*** [7]. These attacks exploit flaws in the *protocol logic* under an active network attacker who does not exploit potential cryptographic primitive weaknesses, called *Dolev-Yao attacker*. Some examples thereof are Machine-in-the-Middle (MITM), authentication bypass, downgrade, replay, credential swapping attacks, etc. The related lines of work have focused on developing formal verification methods and tools aimed at finding or proving the absence of logical attacks. However, formal methods, and in particular program verification, have not solved fundamental problems, such as coping with (combinations of) features that are found in real-world systems such as pointer aliasing or complex control-flow to name a few; thus excluding most *existing*, real-world code bases (such as *OpenSSL*, *wolfSSL*, and *LibreSSL* we will test). For such code bases, those techniques are thus limited to the design level –that is, to the ***protocol specifications*** but not the implementations thereof (*cf* Section 2.3).

Those lines of work have persistently been motivated by the prevalence of logical attacks and the high complexity of manually finding them. Indeed, protocol specifications are often flawed and exposed to ***design-level logical attacks***. For instance, Transport Layer Security (TLS) 1.2 and early drafts of TLS 1.3 suffered from serious design flaws such as complete authentication bypass [24, 13, 22, 23, 15, 42, 16, 4, 31]. In 2017, WPA2 was shown to be entirely broken by a logical attack [53]. The countermeasures implemented by IEEE months later were in turn defeated by another logical attack [54]. Finally, even credit card payment systems suffer from such flaws [9, 8]. Practitioners are now aware of the usefulness of formal methods during protocol design and use them (for example, see IETF calls for help to the formal methods community [43, 56]).

However, specifications are simply an abstraction of the program that end-users deploy and run, and are themselves plagued with frequent implementation bugs. These implementation flaws may introduce additional vulnerabilities, notably ***implementation-level logical attacks***. Examples thereof are abundant and well illustrated by the extensive history of such attacks on TLS: [26, 11, 27, 34, 24, 42, 18, 47]. More generally, even a formally proven specification can result in a flawed and insecure implementation.

*. Part of this work was done when M. Ammann was at Inria Nancy Grand-Est, Université de Lorraine, LORIA, France.

†. A list of changes since the initial version can be found in Appendix A.

Memory-related vulnerabilities and fuzzing. In this paper we are concerned with finding *implementation-level logical attacks* in large cryptographic protocol code bases. For this, we build on *fuzz testing*, which has been developed since the 1990s [41] and is now the gold standard for testing security software. It has become a key part of software development practices; *e.g.* Google [5], Adobe, Cisco, and Microsoft use it at scale on their and others' codebases. Two reasons for the success of fuzzing are its scalability and its extreme efficiency at finding spatial and temporal memory bugs, a class of vulnerabilities that is both notoriously difficult to manually find, and prevalent in today's software. This is generally achieved by using a *fuzzing loop*, in which test cases from a *Corpus* are first randomly mutated (or generated) and then executed against the Program Under Test (PUT). Mutated test cases are then added to the *Corpus* if some *feedback* metric (*e.g.* code-coverage) deems them interesting. When executing test cases, an *objective oracle* observes whether a *security policy* violation occurs (*e.g.* memory corruption). However, even if a protocol implementation is tested against memory-related vulnerabilities, the whole class of ***implementation-level logical attacks*** remains in the blind spot.

Contributions. We answer the lack of effective techniques to preclude logical attacks from protocol implementations with the following contributions.

DY Fuzzing. We propose a novel approach to fuzzing cryptographic protocols dubbed Dolev-Yao model-guided fuzzing (*DY fuzzing* for short). It is based on the novel idea of using formal DY models as domain-specific knowledge to guide the fuzzer and give it the ability to detect logical attacks in protocol implementations.

More precisely, the search space we expose to the *fuzzer mutations* comprises all the *DY traces*. Those represent executions of protocols in the (formal) DY model: networking actions (inputs, outputs) and adversarial manipulations over exchanged messages that are abstracted away by terms and idealized cryptography [7]. We propose new domain-specific **mutations** over DY traces for exploring this search space. Any DY trace can be compiled into a proper *test case* against the PUT through *concretization*: DY terms are evaluated into bitstrings, and DY inputs and outputs are evaluated into communications with agent instances of the PUT.

Next, we define domain-specific **fuzzing security policies and objective oracles**. Indeed, the usual spatial and temporal memory errors-related policies and objective oracle (*e.g.* AddressSanitizer (ASAN) [49]) are unable to detect logical attacks. We still enable them, as they are useful to find memory-related bugs that may be triggered only from specific states that are hardly reachable using standard fuzzing. As we shall see, DY traces are very good at exploring such *deep states* reached only after long or complex executions. This claim is also backed up by four new vulnerabilities we found on *wolfSSL* that were missed by state-of-the-art standard fuzzers. However, if limited to memory related errors, the fuzzer would miss all logical attacks. We solve this by translating formal DY properties that express the absence of logical attacks, *e.g. strong agreement*, into security policies and objective oracles. This way, any violation thereof reached through fuzzing is detected and flagged as attack candidate. Thanks to this, our fuzzer found two known vulnerabilities that are logical attacks and are now detectable.

We stress that a DY fuzzer does not require a complete protocol DY model but only a DY attacker model (by the means of a term algebra and agents the attacker can communicate with) and security policies.

***tlspuffin*: a full-fledged and modular DY fuzzer implementation.** In addition to a generic design specification for DY fuzzers, we also contribute a complete *Rust* implementation of a DY fuzzer for TLS that we applied on three different PUTs: *OpenSSL*, *LibreSSL*, and *wolfSSL*.

Our implementation follows modular design-principles and revolves around three main layers and modules that are of independent interest. First, the protocol- and PUT-agnostic DY fuzzer that we implemented in a standalone module *puffin* uses the main fuzzing loop of the modular, state-of-the art fuzzer *LibAFL* [33]. It implements custom test cases using DY traces, mutations, and objective oracle. On top of *puffin*, we built protocol-dependent fuzzers: *tlspuffin* for TLS and the preliminary *sshpuffin* for SSH. Third, we connected PUTs to the fuzzers: *OpenSSL*, *LibreSSL*, and *wolfSSL* to *tlspuffin* and *libssh* to *sshpuffin*. This FLOSS project of ca. 16k *Rust* LoC is accessible on GitHub [52].

Our fuzzer is fast, it can reach 700 executions per second per CPU core, and *delightfully parallel*. We let *tlspuffin* run on the aforementioned TLS PUTs, which found seven vulnerabilities, including four new CVEs on *wolfSSL*. We shall explain why they were out of the scope of standard fuzzers, either because they are purely logical attacks, which would have been missed with memory-related objective oracles, or only reachable from deep states, which are not or hardly reachable with other standard or other model-based fuzzers.

To summarize, our contributions are as follows.

1) We propose DY Fuzzing: a new approach to fuzzing cryptographic protocols that notably captures for the first time the class of logical attacks. We propose a complete system design of such a fuzzer.
2) We propose *tlspuffin*: a full-fledged, modular, and efficient implementation in *Rust* of this fuzzer design.
3) We evaluate *tlspuffin* on several TLS 1.2 and TLS 1.3 libraries and (re)found seven vulnerabilities, including four new ones (one critical, two high, and one medium).

Outline. We first recall some background about the TLS protocol and fuzzing and discuss related work (Section 2). Then, we provide a DY model that allows us to define the fuzzer's search space. We made this model generic enough in order to then directly link it to the executions of an implementation (Section 3). Building on this generic DY model, we present the concept of DY fuzzing (Section 4) and discuss its implementation in our *tlspuffin* fuzzer (Section 5). We then present the results of fuzzing three TLS implementations and evaluate our tool (Section 6). Finally, we conclude by discussing directions for future work (Section 7).

2. Background

In this section we introduce shortly the TLS protocol, some background on fuzzing and we discuss related work.

2.1. Case Study: The TLS Protocol

The Internet standard TLS [48] is a protocol to establish a secure channel between two agents, a client and a server, communicating over an untrusted network. It is notably widely used in the context of HTTPS and enables web browsers to securely communicate with web servers. TLS aims at providing strong security guarantees. *(i) Authentication*: The server is always authenticated, and the client can be optionally authenticated. Authentication is realized by the means of asymmetric cryptography (*e.g.* RSA) or a symmetric Pre-Shared Key (PSK). *(ii)* Integrity and confidentiality: Application data sent by the agents is always encrypted and integrity-protected with a session key. For simplicity and conciseness, we focus on the latest TLS 1.3 version.

The TLS protocol has two sub-protocols: first, the *handshake* protocol negotiates cipher suites, authenticates the end-points, and establishes a shared, session key; then, the *record layer* protocol uses the established secure channel (based on the session key) to exchange application data. With the aim of being agile and adaptive to the use case, the protocol offers different modes of operation that sometimes can be combined. This yields a rather complex state machine for clients and especially servers [48, Appendix A]. We give a bit more details about the handshake protocol as we will use it for illustration throughout the paper.

Key Exchange and Authentication. The handshake protocol starts with two *key exchange* messages. The client sends a `ClientHello` message containing proposed cipher suites, and either an ephemeral (EC)DH key share or a PSK (or both). The server replies with a `ServerHello` indicating the negotiated connection parameters and, if needed, its ephemeral (EC)DH key share. Alternatively, if the client's proposal does not suit the server (e.g., unsupported cipher suites, or an invalid PSK) the server may send a `HelloRetryRequest`. Any messages after a successful key exchange will be encrypted.

To avoid a MITM attack, if the key was not pre-shared, the server must authenticate. For this it sends a `Certificate`, *e.g.*, a X.509 certificate, and a `CertificateVerify`, that is a signature matching the certified key of the entire handshake. (Optionally the server may request similar messages from the client.)

Finally, both the server and client send a `Finished` message. This message is a MAC over the entire handshake that provides key confirmation and also binds the participants' identities to the session key.

Session resumption. At the end of the handshake the server transmits a `NewSessionTicket`. This ticket may either contain a key identifier, or an encrypted key for the server to allow stateless resumption. This allows for the more efficient PSK mode in a subsequent session as it avoids the certificate based authentication.

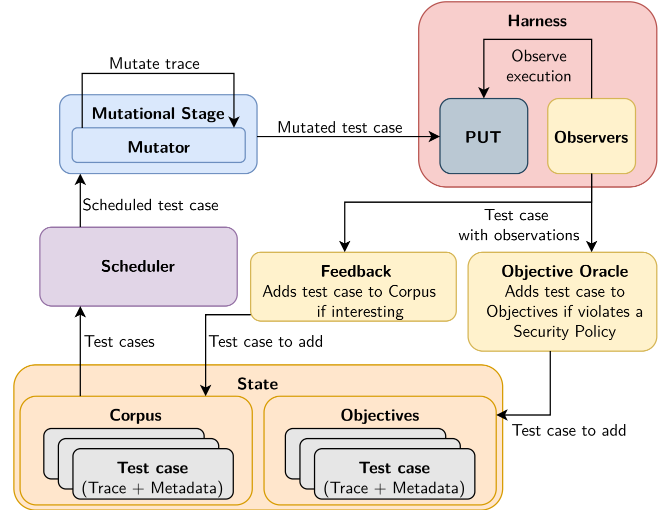


Figure 1: Fuzzer Architecture

TLS Implementations. The most widespread implementation of TLS is *OpenSSL* with an almost 25-year history. *LibreSSL* is a fork thereof and aims to be more secure but has less features. *wolfSSL* is a lightweight implementation widely used by IoT and embedded devices, and is able to run on OSs and CPUs otherwise not supported.

2.2. Fuzzing

One of the gold standards of security-related software testing is *fuzzing* [32, 57, 46, 39], In its general form, *fuzzing* is the action of repeatedly executing the PUT on inputs outside the expected, honest input space to find violations of certain *security policies*. Most of the time, this is done by iterating fuzz runs where new test cases are generated from the current *Corpus*, *i.e.* set of "interesting" test cases, whose execution by the PUT will provide *Feedback* to the algorithm that will guide future test case generation. The nature of the feedback can be diverse: coverage (*e.g. code-coverage*), policy violation, etc.

Different approaches exist for input generation: *generation-based fuzzers* utilize a specification of the input space (*e.g.* a grammar) while *mutation-based fuzzers* leverage (often random) mutations to generate new test cases from the current *Corpus* and add those to this corpus if deemed interesting according to the obtained feedbacks. Finally, an *Objective Oracle* checks *Security Policies* when executing the test cases to detect security violations in the PUT, such as a buffer overflow.

We are mostly using the terminology of *LibAFL* [33], which is a state-of-the-art, modular, efficient fuzzing framework we build on. In its default configuration, *LibAFL* implements *evolutionary fuzzing*, which is mutation-based and grey-box, *i.e.*, it uses some limited PUT runtime information to collect feedback (*e.g.* through code instrumentation). The main *LibAFL* fuzzing loop is depicted in Figure 1.

2.3. Related Work

Fuzzing protocols at the bit-level. Narrowing down the discussion to cryptographic protocols, the main approach is mutation-based fuzzing on the network packets or the inputs of cryptographic primitives [44, 55, 19, 35]. Such testing methodologies are adequate to find safety vulnerabilities with potential security implications[1] but are unable to find logical (*e.g.* MITM) attacks for two fundamental and intrinsic reasons:

(1) logical attack states are not reached: it is overwhelmingly unlikely that random bit flips and basic arithmetic operations on the network packets express valid, logical transformations such as an attacker performing replay, relay, bypass, credential swapping, or downgrade attacks. Indeed, the latter often require numerous, deep, and synchronized modifications in the packets with many bytes modified in a very constrained way at different locations. Moreover, the bit-level representation of test cases (and the code-based notion of coverage) they use fall short of capturing the diversity of executions corresponding to different agents' and attackers' logical actions. This reachability problem also concerns memory-related bugs that are solely reachable from deep states, that we can often reach with a DY attacker.

(2) logical attacks are not detected: the classically used spatial and temporal memory errors-related security policies and objective oracle are unable to detect when a logical attack such as an authentication bypass occurs. That is, even if classical fuzzers reached such an attack state, they would just drop the corresponding test case and deem it uninteresting on the basis that no memory-related error occurred. We revisit and exemplify those gaps in Section 6.1.2.

Cryptographic protocols-oriented fuzzing. Some prior work extended the *stateful fuzzing approach* [6] with generation-based fuzzers whose input space model is some ad-hoc *state machine model* [27, 11, 34, 45]. They are tailored to capture bypass authentication attacks or other violations of the intended state machine flows. We claim that the abstraction level of state machines is too limited to express test cases and mutations as they cannot capture the entire class of logical attacks. For example, they do not allow the attacker to tamper with the message contents, while most logical attacks rely on this. Moreover, since the underlying model is not security-oriented, the security policy violations they find are not necessarily security attacks and require manual inspection (see Section 6.1.2 for more details).

Program verification and secure compilation. We argue in Appendix B that verification methods targeting implementations of cryptographic protocols, such as *F**, *DY**, *Jasmin*, etc., have currently two drawbacks that make them unsuitable to preclude implementation-level logical attacks in *existing* real-world protocol implementations: *scalability* to whole large protocols and, more importantly, ability to *operate on existing, deployed* implementations.

1. *E.g.,* programs may suffer from low-level implementation flaws that do not yield logical attacks such as memory safety bugs (*e.g.* HeartBleed [26] and CloudBleed [25]) for which classical fuzzing can be adequate.

3. The Dolev-Yao Model

Our fuzzing framework builds on the so-called *Dolev-Yao (DY) model*, going back to the seminal work of Dolev and Yao [30] that is today the basis for numerous verification techniques [7] and real-world security analysis [13, 22, 23, 9, 36, 14] of protocol designs (but not of implementations). We now recall some preliminary basic definitions in this model and refer the curious reader to [21, 17].

3.1. Term Algebra

In DY models, messages are described using a term algebra. For example, the term $\mathsf{senc}(m, k)$ represents the message m encrypted using the key k. The algebraic properties of cryptographic functions are specified by equations over terms. For example, $\mathsf{sdec}(\mathsf{senc}(m, k), k) = m$ specifies the expected semantics for symmetric encryption: decryption using the encryption key yields the plaintext. As is common in the DY model, cryptographic messages only satisfy those properties explicitly specified algebraically. This yields the now standard *black-box cryptography assumption*: attackers do not exploit potential weaknesses in cryptographic primitives beyond those explicitly specified. However, as we shall see, attackers will have complete control over the network.

Definition 1 (Terms)**.** *A signature* Σ *is a set of* operators *with their arities. The subset of operators of arity* 0*, is the set of* atoms*. We also assume a countably infinite sets of* variables $\mathcal{V}$*. The set of terms* $\mathcal{T}$ *is defined inductively as the set containing* $\mathcal{V}$ *and terms resulting from applying operators to other terms.*

Intuitively, operators model computations over messages (*e.g.* symmetric encryption: $\mathsf{senc} \in \Sigma$ of arity 2). Atoms model atomic data such as nonces, keys, and constants.

Example 1. *A basic model of digital signatures can be specified by a signature* Σ *that contains operators* $\mathsf{sign}(\cdot, \cdot)$*,* $\mathsf{checksign}(\cdot, \cdot)$*,* $\mathsf{pk}(\cdot)$*, and a constant* $\mathsf{true} \in \Sigma$*.*

Example 2. *As another TLS related example, consider the* `ClientHello` *message. Slightly simplifying (omitting legacy fields and compression methods), we model this message using an operator of arity 3: the term* $ch = \mathsf{CHello}(sid, cs, ext)$ *represents a* `ClientHello` *message where* sid *is a session identifier,* cs *and* ext *are terms that encode the list of proposed cipher suites, respectively extensions. Note that* $\mathsf{CHello}()$ *models the formatting but does not provide any cryptographic protections. We therefore suppose that we also have three operators* π_1, π_2, π_3 *that project each of the arguments to allow an attacker to extract them.*

Remark 1. *Algebraic properties over operators, such as "verifying a signature with a matching key returns* true*", are usually expressed through an equational theory [2]. Continuing Example 1, one would specify the equation* $\mathsf{checksign}(\mathsf{sign}(x, y), \mathsf{pk}(y)) = \mathsf{true}$ *with* $x, y \in \mathcal{V}$*. This does indeed model signature verification as the public key used for verification* $\mathsf{pk}(y)$ *must match the signature key* y*,*

which may be instantiated with any term. To illustrate the black-box cryptography assumption*, we emphasize that this would be the* only *algebraic property satisfied by* checksign*, and hence the* only way *to correctly verify a signature in this model is to use a matching key. This models a strong form of the unforgeability cryptographic assumption.*

In DY protocol verification, an equational theory is required to reason about *protocols [21, 17]. In this work, it is not required as operators and terms will be given a bitstring semantics defined by the concrete implementation of the operators in a cryptographic library (see Section 4).*

3.2. DY Traces

Despite the *black-box cryptography assumption*, attackers have complete control over the network and the exchanged messages: they can eavesdrop on, inject, and tamper with messages. In particular, such attackers can perform MITM, replay, relay, downgrade attacks, etc. Those are examples of *logical attacks*, which are formally defined as the class of attacks that can be expressed by *DY traces* (defined below). The *DY attacker* is an attacker who can perform logical attacks: its behavior is defined as the set of all DY traces.

DY traces are series of networking actions (input and output) a DY attacker can perform. We use the standard notion of *channels* to specify whom the attacker is communicating with. Each channel uniquely identifies an agent. For example, whenever an honest TLS 1.3 client starts a new session it will use a specific channel that the attacker can use to communicate.

Definition 2 (DY trace)**.** *Let $\mathcal{C}$ be a countable set of* channels. *A* DY trace *is a sequence of actions $a_1 \cdots a_n$ such that each action a_i is*

- *either an output* $\mathsf{out}(c, x)$ $(c \in \mathcal{C}, x \in \mathcal{V})$
- *or an input* $\mathsf{in}(c, t)$ $(c \in \mathcal{C}, t \in \mathcal{T})$.

Moreover, if $a_i = \mathsf{in}(c, t)$ and x is a variable in t then there exists a previous output $a_j = \mathsf{out}(c', x)$ $(j < i)$. The set of traces is denoted by $\mathcal{A}$.

Intuitively, the *output action* $\mathsf{out}(c, x)$ indicates that a message, referred to by the variable x, is *output* by c and received by the attacker. The *input action* $\mathsf{in}(c, t)$ indicates that the attacker computes a message and sends it to c, who *inputs* it. This message is obtained by replacing the variables in t by the messages received in the corresponding output actions. Terms in inputs, such as t, are called *attacker terms*[2].

Example 3. *We assume two channels c_1 and s_1 of respectively a TLS 1.3 client and server. Consider the following trace $A_1 \in \mathcal{A}$: $A_1 := \mathsf{out}(c_1, x).\mathsf{in}(s_1, t)$ which corresponds to: (1) the client sends a first message (say) a `ClientHello`) we refer to by x, (2) the term t is then sent by the adversary to the server.*

If $t = x$, then the attacker only forwards the server's response to the client. But the attacker can adopt many attack strategies. We give a few examples. E.g., if $t =$ someError, then the attacker pretends that the client sent some error message. The attacker could also modify the message referred to by x. Suppose that x points to the `ClientHello` ch defined in Example 2. Then $t = \mathsf{CHello}(0, \pi_2(x), \pi_3(x))$ would correspond to the same message but replacing the session identifier sid by the constant 0.

Semantics: Definition. We now present a generic, formal semantics of DY traces that define how they can be executed. These generic semantics can be instantiated into *DY semantics* (as informally discussed in Remark 2 and formally defined *e.g.* in [21, 17]) or into *concrete semantics*, as done in our *DY fuzzing* approach and detailed in Section 4.

Recall that each channel $c \in \mathcal{C}$ corresponds to an honest agent with whom the attacker can communicate. We associate to each channel c the corresponding agent's local *state* s_c and denote by $\mathcal{S}$ the set of all local states. The *global state* is defined by a partial function:

$$\mathsf{s} : \mathcal{C} \rightharpoonup \mathcal{S}$$

which returns this association. We also distinguish the set $\mathcal{S}_0 \subseteq \mathcal{S}$ of *initial* states: when a new agent is created we suppose that it starts in an initial state. We say that a global state s is initial when $\mathsf{s}(c) \in \mathcal{S}_0$ for all $c \in \mathsf{dom}(\mathsf{s})$. The attacker's state is a partial function ϕ that associates the variables x of previous output actions $\mathsf{out}(c, x)$ to an abstract notion of messages M, *i.e.*

$$\phi : \mathcal{V} \rightharpoonup \mathsf{M}.$$

The domain of ϕ contains variables referring to previous outputs from this attacker state. The set of such partial functions ϕ is denoted by Φ.

We suppose a generic specification of honest agents by the means of two abstract partial functions:

$$\begin{array}{ll} \mathsf{output} : & \mathcal{S} \rightharpoonup \mathcal{S} \times \mathsf{M} \\ \mathsf{input} : & \mathcal{S} \times \mathsf{M} \rightharpoonup \mathcal{S}. \end{array}$$

Intuitively, when an agent c is in state s_c, $\mathsf{output}(s_c)$ returns an updated local state s'_c and a message m. Similarly, when a message m is provided to an agent whose local state is s_c, the state is updated to $s'_c := \mathsf{input}(s_c, m)$. Note that those functions are partial as honest agents might block.

In order to transform terms in $\mathcal{T}$ (and notably attacker terms) into messages in M we associate to each operator $\mathsf{f} \in \Sigma$ an *interpretation* $[\![\mathsf{f}]\!] : \mathsf{M}^i \rightharpoonup \mathsf{M}$ when f has arity i. Given $\phi \in \Phi$, we inductively lift $[\![\cdot]\!]$ to terms as follows:

$$\begin{array}{rll} [\![\mathsf{f}(t_1, \ldots, t_i)]\!]_\phi = [\![\mathsf{f}]\!]([\![t_1]\!]_\phi, \ldots, [\![t_i]\!]_\phi) & & \\ [\![x]\!]_\phi = \phi(x) & & \text{if } x \in \mathcal{V} \cap \mathsf{dom}(\phi) \end{array}$$

We are now ready to formally define how a DY trace can be executed, by the means of a transition system for actions, between pairs of a global state s and an attacker state ϕ:

$$(\mathsf{s}, \phi) \xrightarrow{\mathsf{out}(c,x)} (\mathsf{s}[c \mapsto s'], \phi \cup \{x \mapsto m\})$$

when $\mathsf{output}(\mathsf{s}(c)) = (s', m)$ and

$$(\mathsf{s}, \phi) \xrightarrow{\mathsf{in}(c,t)} (\mathsf{s}[c \mapsto s'_c], \phi)$$

when $\mathsf{input}(\mathsf{s}(c), [\![t]\!]_\phi) = s'_c$. Intuitively, an action $\mathsf{out}(c, x)$ updates the state of the agent c and records the message m output by c in the attacker's state ϕ. An input action on the other hand provides a message $m := [\![t]\!]_\phi$ as input to agent c and updates the local state of c. The *attacker's*

2. Attacker terms are often called "recipes" in the literature.

computation of message m is specified by the attacker term t. In particular, the attacker term t may refer to previously output messages using variables in the attacker's state ϕ.

A DY trace $a_1 \cdots a_n$ and an initial global state s_0 define an *execution*

$$(\mathsf{s}_0, \emptyset) \xrightarrow{a_1} (\mathsf{s}_1, \phi_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (\mathsf{s}_n, \phi_n).$$

Remark 2. *As is standard in DY protocol verification, honest agents are usually specified in a formal language such as the applied π-calculus [17] (or multiset rewriting rules [40]). In that case, states in $\mathcal{S}$ correspond to processes. More importantly, the messages* M *are the closed (i.e. without variables) terms –up to the equational theory– with additional private atoms (that model e.g. secret keys of honest participants) and $[\![\cdot]\!]$ is simply the identity function, i.e., the operators are uninterpreted. The formal model yields* output *that defines which (closed) term can be output by a given process and* input *that defines the continuation of the process after inputting a (closed) term.*

In contrast, as we shall see in Section 4, for fuzzing we will define a concrete *semantics. In particular* M *will be the actual packets sent over the network and $[\![\cdot]\!]$ will interpret operators by their actual implementations e.g. in the PUT.*

3.3. DY Security Properties

We use the notion of *claims* to express security properties. Honest agents make such claims to log their state and what they believe to be their environment. *Claims* are expressions $\mathsf{c}(m_1, \dots m_i)$ where c is a symbol with an arity i and $m_1, \dots m_i \in \mathsf{M}$ are messages. We illustrate this notion on two particular kinds of claims that we will use throughout the paper. Let $pk, pk_{peer}, m \in \mathsf{M}$.

- *Agreement claims* $\mathsf{Agr}(pk, pk_{peer}, m)$ express that an agent has public key pk and believes to have agreed with a partner having public key pk_{peer} on data m.
- *Running claims* $\mathsf{Run}(pk, pk_{peer}, m)$ express that an agent has a public key pk and believes to be running a session with a partner having public key pk_{peer} and data m.

For example for TLS 1.3, agreement claims are typically triggered by clients and servers at the end of the handshake while running claims are triggered as soon as m (*e.g.* session identifier) is available. The set of claims is denoted by C. We assume a function $\mathsf{claims} : \mathcal{S} \to \mathcal{P}(\mathsf{C})$ that extracts the set of claims made so far by an agent in a given local state.

Applying the function claims on all local states throughout an execution, we obtain a sequence of sets of claims, called a *trace of claims*. Formally, given an execution

$$(\mathsf{s}_0, \emptyset) \xrightarrow{a_1} \cdots \xrightarrow{a_n} (\mathsf{s}_n, \phi_n)$$

we define the corresponding trace of claims C as $C := C_0, \dots, C_n$ where $C_i = \bigcup_{c \in \mathsf{dom}(\mathsf{s}_i)} \mathsf{claims}(\mathsf{s}_i(c))$ corresponds to all claims extracted from the ith state. Claims are therefore *positioned*, and we say $\mathsf{Agr}(\mathsf{pk}, \mathsf{pk}', m) @ j$ is true in C if $\mathsf{Agr}(\mathsf{pk}, \mathsf{pk}', m) \in C_j$. Our logic for expressing properties is reminiscent to the one used in the Tamarin prover [40].

Definition 3 (DY Properties)**.** *A property is a first-order formula over positioned claims ($c @ i$), equality over messages ($m_1 = m_2$), and comparisons between positions ($i < j$, $i = j$). An execution satisfies a property if it is true on the corresponding trace of claims. A property is true if it is satisfied by all executions defined by all initial states and DY traces.*

Example 4. Non-injective agreement *on some data m is the property:*

$$\forall \mathsf{pk}, \mathsf{pk}', m, i.\ \mathsf{Agr}(\mathsf{pk}, \mathsf{pk}', m) @ i \Rightarrow \exists j.\ \mathsf{Run}(\mathsf{pk}', \mathsf{pk}, m) @ j \wedge j < i$$

Intuitively, whenever an agent (identified by) pk *believes they successfully agreed with agent* pk' *on m then agent* pk' *indeed previously started a session with* pk *on data m.*

4. DY Fuzzing

At a high level, we propose with *DY fuzzing* to use the DY attacker and DY traces as domain-specific knowledge to produce test cases and detect logical attacks.

The **search space**, *i.e.* the set of all *test cases*, is the set of all DY traces $\mathcal{A}$. Starting from a *Seed Corpus*, DY traces are **mutated** and then executed on a PUT. To execute a DY trace, we rely on two components. *(i)* The *Mapper concretizes* terms in $\mathcal{T}$, and notably attacker terms, into bitstrings, *i.e.* it computes $[\![\cdot]\!]$. *(ii)* The *Harness* sends those adversarial bitstrings to the PUT's agent sessions associated to channels in $\mathcal{C}$. The PUT sends back bitstrings that get added to the attacker's state ϕ. Therefore, the set of messages M is the set of bitstrings, and the output and input functions are implemented through the interaction of the *Harness* with the PUT. The *Harness* also observes the execution, extracts claims (*i.e.* function claims), which are then analyzed by the *Objective Oracle* that detects security policy violations, in particular DY property violations.

As is standard, the *Harness* is PUT-dependent. However, we emphasize that the *Mapper* and *Objective Oracle* are only *protocol*-dependent and PUT-independent.

Benefiting from our generic presentation of the DY model in Section 3, we specify those different components in the next subsections. We remain as generic as possible: the idea of DY fuzzing is applicable independently of the protocol and the PUT, provided that it does implement a cryptographic protocol. We exemplify some aspects with our main implementation, *tlspuffin*, that implements a DY fuzzer for TLS against various TLS libraries (see Section 5).

4.1. Mapper

The *Mapper* can use the PUT's implementation of a primitive f to compute the concretization $[\![\mathsf{f}]\!]$. However, a different implementation or an external library could also be used. For instance in *tlspuffin*, we reused part of the Rust library *rustls* that implements TLS to compute $[\![\cdot]\!]$ for the 189 symbols of the signature we used. In the rest of the document, the cryptographic protocol primitive's implementation used by the *Mapper* is referred to as *SignatureLib*. It can be a reference implementation (*e.g. rustls*), the PUT itself,

or a from-scratch implementation. When multiple PUTs implementing the same protocol are tested, the same *Mapper* can be reused. Hence, the *Mapper* is PUT-independent and can be written once per protocol; as we did for *tlspuffin*.

As mentioned, we let M be the set of bitstrings. We make sure to use an unequivocal representation of data as bitstrings; *e.g.* we use the DER format for certificates. For $\mathsf{f} \in \Sigma$, the *Mapper* uses the corresponding function in *SignatureLib* implementing this function for computing $[\![\mathsf{f}]\!]$. $[\![\mathsf{f}]\!]$ is a partial function since its computation might fail or return an error. For an atom f_0, we define $[\![\mathsf{f}_0]\!]$ as the corresponding, statically generated, data item. For instance in *tlspuffin*, we statically generate the RSA public keys for a bounded number of agents and then bind each to a constant in the term algebra. Some of those agents are assumed to be compromised (in control of the adversary). For those, we also include the corresponding RSA private key in the term algebra so that the attacker can use them in attacker terms (in inputs).

4.2. Harness

As is standard with fuzzing, a PUT-specific *Harness* is required. While it could be possible to create a *Harness* that just passes a single protocol message to an agent, it would fail to reach parts of the code only accessible after several flights of messages and potential local state updates.

DY fuzzing neatly addresses those challenges with a *Harness* that executes DY traces on the PUT as explained next. Given a test case, that is a DY trace $A = a_1 \cdots a_n \in \mathcal{A}$, it will do the following.

Agents creation. The *Harness* creates a new PUT session s_c for each channel $c \in \mathcal{C}$ that appears in A. The state of each of these sessions then corresponds to the local state of the agent identified by c in our generic DY model. Those sessions have a notion of *input buffer* and *output buffer*, where bitstrings can be read, respectively written, as well as a notion of *progress*: we assume a function *Progress* can be called on a session to instruct the corresponding agent to read and process a message on the input buffer and possibly write a message on the output buffer. Configurations of those sessions, and thus their initial states $s_c \in \mathcal{S}_0$, are those of the channels (*e.g.* client vs. server). The initial global state s_0 is composed of all those s_c.

For example for the *OpenSSL* implementation of TLS 1.3, the *Harness* creates new SSL objects (pointers of type SSL$*$ storing a server or client session state) for each channel in A. The *Progress* function is **int** SSL_do_handshake(SSL $*$ssl).

Communication. The actions of the trace A are executed according to the transition relation $\xrightarrow{a}$ (see Section 3) instantiated as follows. First, $[\![\cdot]\!]$ is computed by the *Mapper* (see Section 4.1) on attacker terms. Second, we need to instantiate the input and output functions. The state $s'_c := \mathsf{input}(s_c, m)$ is obtained by first writing m into the input buffer of the current state s_c and then calling the *Progress* function that modifies the state to s'_c. $(s'_c, m) := \mathsf{output}(s_c)$ is computed by reading m from the output buffer of the current state s_c; the resulting state s'_c is identical to s_c up to the output buffer.

Claim extraction. The *Harness* is also responsible for extracting claims from agents, that is it implements a function $\mathsf{claims} : \mathcal{S} \to \mathcal{P}(\mathsf{C})$ (see Section 3.3). In general, doing so may depend on the PUT and the possibility to introspect the agents' internals but is often straightforward to do because required data are usually exposed. We detail two different approaches to do so in Section 5.2 we used for *tlspuffin*. If the PUT offers no or no easy introspection (*e.g.* the PUT is closed-source), it is still possible to extract some claims solely based on the DY trace being executed and possibly the exchanged messages by interpreting parts of the exchanged messages.

4.3. Seed Corpus

We consider a bounded number of agents (and thus of channels), enough to be able to express all possible *happy flows*[3] for all possible expected protocol configurations. For obvious reasons, our fuzzer evaluations solely use happy flows and no attack trace as seeds.

For instance for TLS 1.3, we consider two different honest agents, Alice and Bob. We consider seeds for each of the following happy flows: *(i)* full handshake between Alice and Bob without decrypting messages (DY attacker acting as a passive MITM), *(ii)* full handshake between the DY attacker, acting as an honest server, and Alice acting as client, *(iii)* full handshake between the DY attacker, acting as an honest client, and Bob acting as server, and *(iv)* the happy flow *(iii)*, followed by a resumption handshake with the same agents. Note that additional scenarios could be considered, but a good balance has to be found between the size of the search space and the complexity and the number of the seed scenarios.

4.4. Mutations

Since the *Seed Corpus* captures all relevant execution scenarios for a given fuzzing campaign, we only consider mutations that do not create new channels. However, mutations can modify the structure of the trace at the *action-level* (*e.g.* swapping two actions, dropping a message, etc.). They can also modify the content of the attacker terms at the *term-level* (*e.g.* replace a sub-term by another one to express a credential swapping, add a sub-term to add a TLS 1.3 extension that was inexistent, etc.). The mutations we consider for *tlspuffin* are described in Table 1 (split into action- and term-level mutations). We consider these to be a good basis for any DY fuzzer as they are completely protocol (and obviously PUT) independent.

4.4.1. Action-level mutations. The *Skip* mutation removes a random action from a trace. The *Repeat* mutation repeats

3. We call *happy flow* an honest and expected message flow. If the adversary is a MITM, then it forwards messages without modifying them. If it acts as a server or client, it behaves as an honest one.

Mutation	Description
Skip	Removes an action from a trace
Repeat	Repeats an action from the trace to the trace
Swap	Swaps two (sub-)terms in the trace
Generate	Replaces a term by a random one
Replace-Match	Swaps two operators in the trace
Replace-Reuse	Replace a (sub-)term by another (sub-)term in the trace
Remove-and-Lift	Replaces a (sub-)term by one of its sub-terms

TABLE 1: Mutations

an action: a random action in the trace is copied and inserted at a random new position (and in case of an output actions, the output variable is renamed). These two action-level mutations are already enough to capture some authentication bypasses such as the ones from [11] and **SDOS** from Table 2.

4.4.2. Term-level Mutations. A major advantage of DY fuzzers is their ability to also mutate attacker terms and thus deeply change the structure of exchanged messages (*e.g.* replace, remove or add TLS extensions) and/or modify very specific fields (*i.e.* for expressing a certificate swapping). These mutators require a description of test cases that specifies the structure of messages, which is one of the main novelty of our DY fuzzing approach. Implicitly, they all start by randomly picking an input action $\mathsf{in}(c,t)$ and then mutate the attacker term t. For example, the *Replace-Match* mutation replaces an operator $f \in \Sigma$ in t with a different one $f' \in \Sigma$ of the same arity as f. This can be seen as changing the implementation of some computations (*e.g.* replacing SHA2 with SHA3) or changing the values of some constants (*e.g.* swapping Alice's public key with Bob's public key). Another example is the *Remove-and-Lift* which chooses at random a subterm t' of t and replaces it with a random sub-term t'' of t'. In particular, this mutation allows to remove random elements in a list.

Example 5. *Suppose we have an initial trace* $A = \mathsf{in}(s_1, ch).\mathsf{out}(s_1, x)$ *where* $ch = \mathsf{CHello}(sid, cs, ext)$ *as in Example 2. This models the case where a server receives a* `ClientHello` *from an attacking client before responding with a term* t*. Suppose that* cs *and* ext *are* `nil` *terminated lists and contain, respectively a single cipher suite* `c` *and a list of* n *extensions* $e_1, \ldots, e_n$*. Then, the attacker can*

- *remove extension* e_n *by* $ch_1 :=$ *Remove-and-Lift*(ch) $= \mathsf{CHello}(sid, [\mathtt{c}, \mathtt{nil}], [e_1, \cdots, e_{n-1}, \mathtt{nil}])$,
- *add a cipher suite* `c` *by* $ch_2 :=$ *Replace-Reuse*(ch_1) $= \mathsf{CHello}(sid, [\mathtt{c}, \mathtt{c}, \mathtt{nil}], [e_1, \cdots, e_{n-1}, \mathtt{nil}])$,
- *iterate Replace-Reuse* k *times and obtain* $ch_k := \mathsf{CHello}(sid, [\mathtt{c}, \cdots, \mathtt{c}, \mathtt{nil}], [e_1, \cdots, e_{n-1}, \mathtt{nil}])$ *where* `c` *is repeated* k *times.*

Sending ch_k *to the server may actually trigger a* `HelloRetryRequest` *message due to the missing extension, e.g. removing the* `supported_groups` *extension. Then applying the Repeat mutation, we obtain the trace* $A' = \mathsf{in}(s_1, ch_k).\mathsf{in}(s_1, ch_k).\mathsf{out}(s_1, x)$*. As we will see in Section 6 this trace actually leads to a buffer overflow on wolfSSL when* $13 \leq k \leq 150$.

4.4.3. Mutation Constraints. Mutations can be applied only when the following constraints are fulfilled.

Well-typed preservation. Consider *Swap* that replaces an operator f by f'. We wrote that the arity of f and f' should match. In our presentation, messages are un-typed (they are simply elements of M). But in practice, most programming languages of the *SignatureLib* (or PUT) have type constraints and the implementations of f and f' are likely to be typed in a more fine-grained manner. For instance, an argument of f could be given the PUT-specific type for X.509 certificates. Hence, we impose that f and f' have the same type in order for the mutated term to still be well-typed. More generally, the application of mutations should preserve that the trace is well-typed. Moreover, we skip any mutation that implies a term computation (through $[\![\cdot]\!]$) to fail.

Size bounds. There is a special case in which mutations can also be skipped. If the trace length or an attacker term is becoming too big or small, then the mutation is skipped. The reason for this is that we are not interested in indefinitely creating larger and larger traces. There are sane limits for trace lengths and attacker terms sizes which can be determined by observing how big the defined seeds are.

4.5. Security Policies and Objective Oracle

The *Objective Oracle* observes executions made by the *Harness* and looks for security policy violations. When such a violation is detected, the corresponding test case is flagged as an objective and is stored to disk.

Memory-related *Objective Oracle*. Fuzzing has mostly been used to discover memory related bugs. Those bugs are easy to detect by relying on operating system signals or code instrumentation techniques such as ASAN [49]. DY fuzzers can use those but we strive to go beyond since, as is, the fuzzer would silently miss logical attacks.

DY properties as policies. To remedy this problem, we consider DY properties (Definition 3) as additional security policies. For example, for TLS 1.3 and *tlspuffin*, we consider the DY properties corresponding to agreement on handshake data as policies in addition to using ASAN. We next explain how one can translate DY properties into an operational *Objective Oracle* detecting them.

***Objective Oracle* for DY properties.** As explained in Section 4.2, the *Harness* provides a function $\mathsf{claims} : \mathcal{S} \to \mathcal{P}(\mathsf{C})$. Therefore, a trace of claims can be extracted throughout the execution by the *Harness* and the *Objective Oracle* can evaluate the validity of all the considered DY properties at any execution step, according to the properties semantics defined in Definition 3. As soon as one DY property is falsified, the *Objective Oracle* flags the corresponding test case as an attack candidate.

4.6. Big Picture: The DY Fuzzing Loop

Each fuzzing campaign starts with a *Corpus* initialized to the *Seed Corpus*. The main, standard fuzzing loop proceeds as follows: a DY trace $A \in \mathcal{A}$ is picked from the current Corpus, multiple mutations are applied yielding $A' \in \mathcal{A}$.

The *Harness* executes A' on the PUT and, when necessary, calls the *Mapper* to concretize all attacker terms in A''s input actions. It also collects feedback (*e.g.* code-coverage in the PUT through code instrumentation) and observations (claims and potential ASAN errors). If the *Objective Oracle* identifies a security policy violation on those observations, notably a DY property violation, it flags A' as an attack trace and stores it. Otherwise, based on the obtained feedback, the fuzzer decides whether A' is worth being stored in the *Corpus*. The fuzzer then proceeds to the next loop iteration.

Finally, the user can inspect the found attack traces $A_{\mathsf{attack}} \in \mathcal{A}$, which unambiguously and clearly describe an adversarial behavior of the DY attacker. We provide such traces and graphical representations thereof found and produced with *tlspuffin* in [52].

5. Implementation: The *tlspuffin* Fuzzer

We present *puffin* and its derivatives, which altogether is a public FLOSS project hosted on Github [52] written in *Rust*. We notably contribute a generic *Rust* library for building DY fuzzers (*puffin*) for arbitrary protocols and a full-fledged DY fuzzer for TLS (*tlspuffin*) using *puffin*. We used *tlspuffin* to fuzz *OpenSSL*, *LibreSSL*, and *wolfSSL*. To show the modularity and generality of *puffin*, we also briefly mention *sshpuffin*, which is a preliminary DY fuzzer for SSH that also uses *puffin*.

We wrote for this project ca. 16k *Rust* LoC (computed with `cloc`, excluding dependencies): 6k for *puffin*, 8k for *tlspuffin*, and 2k for *sshpuffin*. As part of *tlspuffin*, the *Harness* for *OpenSSL*, *LibreSSL*, and *wolfSSL* are written in respectively 577 LoC and 500 LoC. Overall, *puffin* and *tlspuffin* took four person-months and *sshpuffin* took two person-months.

5.1. Modular Architecture

Our project features a modular design that facilitates reuses. We present its three main modules (aka *Rust crates*).

puffin is a generic *Rust* library to build DY fuzzers. It is protocol- and PUT-agnostic: it defines a minimal interface (aka *traits*) for a protocol and its security properties as well as for the PUT-harness. Given a crate implementing this interface (*e.g. tlspuffin* for TLS and the 3 aforementioned PUTs), *puffin* implements a DY fuzzer for them.

puffin builds on the state-of-the-art *LibAFL* [33] fuzzer in order to implement an *evolutionary* fuzzing loop. Leveraging the modularity of *LibAFL*, we implement custom and domain-specific *Harness*, *Mutator*, and *Objective Oracle*, as well as an additional *Mapper* component. For this, we implement a generic term algebra amenable to the Fuzzing setting: terms must be serializable, executable (*i.e. Mapper*), introspectable for the mutations (see Section 5.2), etc. Similarly, we implement generic DY traces that can be executed on any given PUT (*Harness*). Since the traces of the *Seed Corpus* must be written by hand, we made *puffin* offer a Domain Specific Language (DSL) for DY traces for declaratively defining traces (more on this in Appendix C.1). We use the standard AFL-like code edge coverage map [33] (*i.e.* hit counts) as feedback metric.

We implemented all mutations from Table 1 resulting in a generic *Mutator*. Finally, all given DY properties are checked throughout the execution by our custom *Objective Oracle*. Each protocol and PUT given to *puffin* are provided as crates which implement some required traits.

Using *puffin*, one can launch a fuzzing campaign on a given PUT, which is the main use case, but also execute a given trace (*e.g.* an objective, or a custom trace written with our DSL) on a given PUT, produce a graphical representation of a trace, or compute the packets agents send in a trace (using $[\![\cdot]\!]$ and the *Mapper*). We implemented all those features, which considerably improve the workflow, as we shall see in Section 5.3.

tlspuffin is a crate that implements for *puffin* the protocol and properties description of TLS. It re-uses parts of the *rustls* crate, that implements TLS in *Rust*, to build a *Mapper* for TLS and its 189 operators. *tlspuffin* is shipped with *Harnesses* for *OpenSSL*, *LibreSSL*, and *wolfSSL*, which can be fuzzed out-of-the-box. The *Rust* Cargo build system offers support for compiling and linking with arbitrary projects, which eases the integration of new PUT code bases to wrap them in modules providing the required traits (the *Harness* for *OpenSSL*/*LibreSSL* and for *wolfSSL* respectively required ca. 500 and 577 *Rust* LoC). *tlspuffin* has a command line interface to enable its different features, choose the PUT and its version, etc. (see documentation in [52]).

sshpuffin is preliminary work demonstrating that one can easily use *puffin* for other protocols, here the SSH protocol and *libssh* as PUT. Except when said otherwise, we focus on *tlspuffin* and *puffin* in the rest of the paper.

5.2. Implementation Challenges

We now review some challenges we tackled in building *puffin* and *tlspuffin*.

Performance. Performance was a key criterion to make implementation choices. For instance, we chose to implement the communication between the agents through in-memory buffers, rather than relying on *e.g.* TCP (which we offer as an additional feature though). We also chose *LibAFL* for its high performance notably due to its multiprocessing capabilities. Multiple fuzzer clients can be running on different CPU cores and share the same *Corpus*. As a result *tlspuffin* allows *delightful parallelism*. For this, we had to make many components serializable: the term algebra and its link with $[\![\cdot]\!]$ (how to concretize), DY traces, claims, etc.

As we shall see in Section 6.2.5, those features make our DY fuzzer scalable and blazingly fast.

Gathering Knowledge from Protocol Outputs. Let us recall that when executing an action $\mathsf{out}(c, x)$, the message $m \in \mathsf{M}$ from $(s'_c, m) = \mathsf{output}(s_c)$ gets added to the attacker's state ϕ by assigning m to the variable x (see Section 3.2). Moreover, the message $m \in \mathsf{M}$ is a bitstring, not a term $t \in \mathcal{T}$. This is not a problem in theory, as the attacker can construct adversary terms using functions to

e.g. access fields in m (see projections from Example 2). This way, he can treat m as a term. However, this would yield quite large attacker terms, since any reuse of a field in m would possibly already require several operators. To mitigate this and simplify attacker terms, we decided that *puffin* would partially interpret m by extracting all sub-messages that can be accessed in plaintext. All those sub-messages m_i are assigned variables x_i, which are added to the attacker's state ϕ and that are thus made available to the attacker. We stress that this does not change the executions and the attacker's behaviors that can be explored. For example, from a `ClientHello` like in Example 2, *tlspuffin* automatically extract as sub-messages the bitstring associated to the TLS version, client random, session identifier, list of cipher suites, and list of extensions.

Queries. We also tackled another challenge related to the way the attacker refers to its state ϕ. To illustrate the problem, consider a variable x and an attacker term $t = \mathsf{sdec}(x, k)$ (assuming k is mapped to a symmetric key) in a trace $T = T_1.\mathsf{out}(c, x).\mathsf{in}(c', t)$ included in the *Seed Corpus*. When executing this trace T, x is assigned a message that can indeed be decrypted with k. Now, after some mutations affecting T_1, the agent c might not send an encrypted message anymore but *e.g.* an error message. Yet, the attacker term t remains the same and its computation will now fail. The problem is that the way the attacker accesses resources that were gathered throughout execution is not robust enough through successive mutations. We alleviate this issue with *queries*. When adding some (sub-)message m to ϕ, *puffin* and *tlspuffin* also stores from which agent $c \in \mathcal{C}$ it originates, the kind of message it is (for TLS: `ClientHello`, `Finished`, etc.), and its internal *Rust* type (accessed though compile-time reflection). A query is simply a conjunction of conditions over metadata (c, message kind, message type). When applied on an attacker's state ϕ, it returns the first matching message. The attacker can use *queries* to access its knowledge in place of variables in attacker terms and traces. It also eases the writing of the seed traces.

Claim Extraction. *puffin* and *tlspuffin* offer different methods for extracting claims from PUTs. One, we used for *wolfSSL*, is to leverage existing potential callback facilities in the PUT to expose the agent's context from which the required data can be retrieved. Another method we used for *OpenSSL* and *LibreSSL*, is to create a minimal C interface of data the PUT must expose for *tlspuffin* to be able to extract the required data. We implemented this in a dedicated *tlspuffin-claims* crate. When using this method, the PUT needs a (lightweight) patch to meet this interface.

Transcript Extraction. When running earlier versions of *tlspuffin*, we noticed that attacker terms could get very large for the trace to be executed gracefully (sometimes with >10k operators). Our investigations have shown that transcript hashes in TLS 1.3 dramatically contributed to this problem. A transcript hash is a hash value over all previously sent and received messages and is included in all authentication messages (*e.g.* `Finished`). Therefore, when a mutation modifies a field in an exchanged message somewhere, other mutations should also be applied to reflect this modification on all next transcript hashes. This dramatically reduces the likelihood of finding mutations that also mutate the transcript hash accordingly. We decided to give the attacker the possibility to use in attacker terms a shortcut $\mathtt{hashTr}\,@\,c$ that refers to the hash transcript as c would compute it at this time of the execution. This is without loss of generality since the transcript being hashed is made of messages sent in clear-text, so the attacker knows it already. We are using the same implementation methods that we use for claim extraction for transcript extraction.

5.3. Other Features

Triaging Bugs. Once objectives are found, *i.e.* traces triggering security policy violations, one needs to triage those. With *puffin*, it is possible to execute traces which are stored on-disk against any PUT, using its dedicated *Mapper*. For better understanding and reproducibility of the bugs, we made *tlspuffin* offer an easy way to also test a trace against arbitrary applications. Indeed, *tlspuffin* is capable of executing a given trace (or even a fuzzing campaign, which will be slower though) against arbitrary TCP clients or servers (on a given address and port), which can serve as Proof-of-Concept (PoC). This is also useful to test against closed-source binaries, or remote servers. Finally, if the bug does not require more than 1 flight of message, say m, then *puffin* and *tlspuffin* also offer a feature to compute the bitstring $[\![m]\!]$ which can serve as a minimal PoC. Those bitstrings are supposed to be sent to a TCP client or server through standard Linux tools like `netcat`.

When we wanted to fully understand the bugs we found, we followed a methodology that we illustrate on a case study in Appendix D.4 and that takes advantage of all the aforementioned features as well as standard debugging tools.

Regression Testing. The *puffin* and *tlspuffin* tools can also be used for other tasks beyond fuzzing. We are already using our tools to test for regressions in the supported PUTs. We treat the attack traces for various vulnerabilities as regression tests, which ensure that bugs which are already known will never occur again.

Analysis Framework. *tlspuffin* also offer features to allow developers to analyze TLS libraries in the flavor of *TLS-Attacker* [50] (more on this in Appendix D.3).

6. Results and Evaluation

In the following, we present the vulnerabilities *tlspuffin* found, including new ones (see Table 2). We explain why our DY fuzzer approach was key and often necessary to find these vulnerabilities. We then provide a detailed evaluation of our tool.

Benchmark Suite. The first step to evaluate our work was to establish a benchmark suite. The Magma project [37] proposes a consolidated benchmark suite of 20 memory-safety related bugs in *OpenSSL*. DY fuzzers could not be satisfyingly evaluated on such benchmarks since they aim at finding a different class of attacks (*i.e.* logical attacks). Therefore, we first selected recent and known logical

CVE ID	AKA	CVSS	Type	New	Version	TLS
2021-3449	**SDOS1**	5.9	DoS	✗	1.1.1j	1.2
2022-25638	**SIG**	6.5	Auth. Bypass	✗	5.1.0	1.3
2022-25640	**SKIP**	7.5	Auth. Bypass	✗	5.1.0	1.3
2022-38152	**SDOS2**	7.5	Server DoS	✓	5.4.0	1.3
2022-38153	**CDOS**	5.9	Client DoS	✓	5.3.0	1.2
2022-39173	**BUF**	7.5	DoS	✓	5.5.0	1.3
2022-42905	**HEAP**	9.1	Info. Leak	✓	5.5.0	1.3

TABLE 2: (Re)discovered vulnerabilities with *tlspuffin*. The CVSS scores are the severity scores attributed by NIST. The "New" column indicates whether the vulnerability was first discovered using the *tlspuffin* tool (✓) or rediscovered (✗). **SDOS1** affects *OpenSSL*. The others affect *wolfSSL*.

attacks: the three first bugs from Table 2 on *OpenSSL* and *wolfSSL*. From this initial ground-truth seed of known bugs, we then added the newly discovered bugs. The result is a first relevant benchmark suite of logical attacks, which we plan to augment in the future.

6.1. Results

After reviewing the (re)discovered bugs in detail, we argue that DY fuzzers are superior at finding them.

6.1.1. Succinct Vulnerability Descriptions. [4] The **SDOS1** vulnerability allows malicious clients to crash *OpenSSL* servers during TLS 1.2 renegotiation by omitting the `signature_algorithms` extension but including a `signature_algorithms_cert` extension.

SIG and **SKIP** are two bugs allowing client authentication bypass in *wolfSSL* servers [47]. A malicious client triggers **SIG** by introducing a mismatch between the signature algorithm in the `Certificate` and `CertificateVerify` messages, which will cause the server to accept any certificate. For triggering **SKIP**, the client just skips the `CertificateVerify` message and achieves the same bypass. Thanks to the capability of *tlspuffin* to create such logical modifications in messages and to automatically detect an authentication bypass through its *Objective Oracle*, it found these two vulnerabilities.

SDOS2 allows clients to crash *wolfSSL* TLS servers. When a TLS 1.3 client resumes a previous session that has been cleared with the *OpenSSL* compatibility layer of *wolfSSL*, then the server crashes with a segmentation fault. *tlspuffin* considers traces that can create multiple sessions and resume them at will and thus found this vulnerability.

CDOS is a bug allowing servers or MITM to crash *wolfSSL* clients. An attacker triggers this bug by sending a large `NewSessionTicket` message (> 256 bytes) to a client with a non-empty session cache, who will then free a pointer to non-allocated memory and crash.

BUF is a stack buffer overflow in *wolfSSL* servers with the *potential* for Remote Code Execution (RCE). Malicious clients can cause a buffer overflow by sending specific `ClientHello` messages to servers: the list of cipher suites they offer must contain *duplicate* ciphers (at least 13); they should pretend to resume a previous session with the appropriate extensions; and they should omit the `supported_groups` extension. The trace from Example 5 can be mutated further to produce such an attacking trace that triggers **BUF**. We explain in Appendix D.4 why such a trace triggers the bug, its root causes, and how we proceeded to obtain such data. The triggerable stack buffer overflow has an attacker-controlled length with a maximum of 44700 bytes. Therefore, large portions of the stack can get overwritten, including return addresses. This vulnerability has the (unconfirmed) potential for an exploit that triggers misbehavior (through stack rewrites) or RCE.

HEAP is a heap buffer over-read bug on *wolfSSL* servers using the `WOLFSSL_CALLBACKS` feature flag, which was meant for debugging but not discouraged for production. The bug can be triggered by sending to a server a maliciously crafted `ClientHello` message with about a dozen `key_share` extensions.

6.1.2. Advantages of DY Fuzzing. *tlspuffin* found all of these vulnerabilities, including four new ones that were missed by the intensive fuzzing effort on *wolfSSL*. Having detailed the reasons why standard fuzzers are unsuitable for finding such vulnerabilities in Section 2.3, we now come back to them in light of these examples. We also explain why DY Fuzzing is in a sweet spot for finding them.

Reachability. The first problem is *reachability*: Can the vulnerability even be reached? *Harnesses* of state-of-the-art bit-level fuzzers [32, 57, 46] usually send only one message to the PUT.[5] This excludes vulnerabilities that require at least one round trip, such as all vulnerabilities from Table 2 except for **HEAP**.

Next, there is the problem of the choice of mutations. Standard fuzzers rely on bit-level mutations: because the sent bitstrings are modified at the bit-level, it is overwhelmingly unlikely that logical transformations of messages will be discovered (*e.g.* at the term level). As an illustration, consider the attacker term $\mathsf{senc}(\mathsf{sdec}(x, k), k')$, which expresses that the attacker, instead of forwarding x, decrypts and re-encrypts x with a different key. With bit-level mutations, the probability for this mutation to happen is upper-bounded by the probability of breaking the encryption scheme. With DY Fuzzing, this adversarial behavior is obtained with a few mutations. The vulnerabilities from our benchmarks (except **HEAP**) rely on rather complex attacker terms obtained by a simple series of mutations (say x is mutated into t) such that the obtained bitstring $[\![t]\!]_\phi$ is very unlikely to be reached by bit-level mutations from $[\![x]\!]_\phi$.

Even **HEAP**, whose only input action contains a simple attacker term, which is *theoretically reachable and in scope*

4. We provide the traces found by the tool that trigger those vulnerabilities and comprehensive vulnerability reports (for ours) in [52].

5. See for example this *OpenSSL* example: https://github.com/openssl/openssl/blob/master/fuzz/client.c

by standard fuzzing (triggerable with a simple attacker term and detectable with ASAN), was not found *in practice* by the large-scale oss-fuzz[6] project continuously run by Google (*wolfSSL* is one of their targets) or the *wolfSSL* foundation itself, which runs seven fuzzers internally every night[7].

Therefore, more-structured test-cases are needed, and the DY setting — which captures logical, adversarial behaviors — provides the exact model needed.

Detection. Classical fuzzers primarily aim at finding memory-related bugs, but their *Objective Oracle* is unable to detect logical attacks. In practice, even if they reached the bug and triggered a logical attack, such as the authentication bypass **SIG** and **SKIP**, the aforementioned state-of-the-art bit-level fuzzers would actually drop this test-case as uninteresting because they would not realize a policy violation occurred.

Again, a more structured approach is required, where a notion of session, agent (here channel), and claims are available to the oracle, as done in our DY Fuzzing approach.

On State-Machine-Guided Fuzzers. We compared our work with related structured fuzzers or testing engines in Section 2.3. We now discuss them in light of our results. **SIG** and **SKIP** were found by a state-machine learner [47], which is not a fuzzer but a different approach capable of finding *some logical* attacks. More precisely, it captures adversarial behaviors that consist of dropping or repeating whole Handshake messages without the ability to modify their contents, except for a few hard-coded pre-defined messages that can be used. In particular, as is [47] could not reach **CDOS**, **BUF**, and **HEAP**. Moreover, such techniques do not feature an *Objective Oracle*; the expected output is a graph of execution flows that need to be manually interpreted to detect potential attacks. Similar conclusions apply to the fuzzer [11].

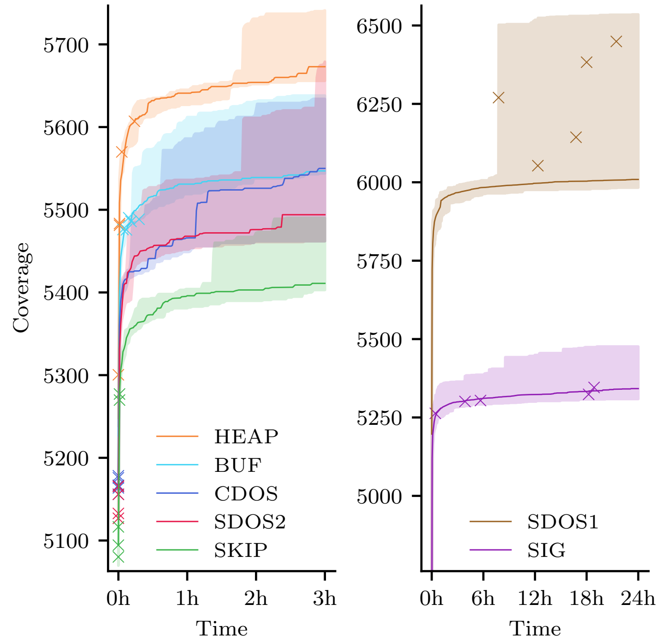
6.2. Evaluation

We faced open **research questions** regarding our first *tlspuffin* implementation. How reliable is our tool at finding bugs? When our tool does not reliably find a bug, what are the reasons for this?

To answer these questions, we conducted a quantitative and qualitative evaluation of our benchmarks, *i.e.* bugs from Table 2. The quantitative results are derived from statistics gathered during the fuzzer execution and by observing mutations. We qualitatively evaluated the fuzzer corpora after the execution of the fuzzer by analyzing their coverage.

6.2.1. Methodology. The evaluation was conducted on a server with 500 GB of RAM and 2 `AMD EPYC 7F521` processors with 16 physical cores each (for a total of 64 logical cores). (Note that such a large computation power and amount of RAM are not mandatory to use *tlspuffin*.) The experiments conducted in this evaluation are completely reproducible. We created bash scripts [52], which download statically linked binaries from our CI and execute experiments.

6. https://google.github.io/oss-fuzz/
7. https://www.wolfssl.com/fuzz-testing/



(a) Group A: 5 experiments per vuln., each on 12 cores

(b) Group B: 90 experiments per vuln., each on 1 core

Figure 2: Code edge coverage and discovery times for vulnerabilities from Table 2. The crosses indicate discovery times for the vulnerabilities. We depict with the shade area the interval over multiple experiments. The order of the vulnerabilities in the legend matches the order in the graph.

We used the following PUTs: *wolfSSL* 5.1.0, 5.3.0, 5.4.0, and *OpenSSL* 1.1.1j. For each PUT and vulnerability from Table 2 that affects it, we designed a specific experiment to evaluate the ability of the fuzzer to find it. That is, we applied on the PUT all the patches for all listed vulnerabilities except the chosen one. We created two groups of vulnerabilities: Group A (listed in Figure 2a) are those found in a matter of minutes, and group B (listed in Figure 2b) are those found in hours. We executed each of the experiments in group A 5 times for 3h on 12 cores each. To mitigate the high variance for group B, those experiments were executed 90 times for 24h on 1 core each.

6.2.2. Experiment Results. In Figure 2a (Group A) and 2b (Group B), we present the code edge coverage and vulnerability discovery times.

Out of a total of 180 experiments in group B, we discovered SIG and SDOS1 only five times each. In group A, all bugs were *always* found within minutes after launching the campaign, whereas in group B discovering the bugs took hours. The mean time until SIG was discovered was 564min (± 510min). For SDOS1, the mean time was 915min (± 318min). For all experiments in group A, the mean time until the bug is discovered was 9min with a maximum standard deviation of 5min. In the following sections, we will discuss the reasons for this variance by first reviewing quantitative results like discovery durations, mutation performance and

fuzzer performance, and then discussing the fuzzer feedback on a qualitative basis by analyzing corpora.

6.2.3. Reachability from the Mutations. We evaluated that the proposed mutations from Section 4.4 are indeed capable of reaching the **SIG** and **SDOS1** vulnerabilities. This does not mean that the series of mutations needed to reach them is likely to happen. For measuring this, we designed a test that counts how many random mutation-tries are required in order to reach a desired test case (*e.g.* one that triggers **SIG** or **SDOS1**). In order to mutate a happy flow trace into a trace that triggers **SIG**, we need at least three mutations in a specific order: twice *Replace-Match*, and then *Generate*. When conducting the experiment over 100 times, on average 809 mutation tries are required to reach **SIG**. For **SDOS1**, on average 3053 mutation tries are required. From those numbers, we can conclude that the vulnerabilities are indeed reachable by the implemented mutations. (Note that the total number of executions in a fuzzing campaign can easily reach the 10^8 range on powerful hardware.)

6.2.4. Corpus & Feedback Analysis. We now answer a natural question: why did only 10 out of 180 experiments find **SIG** and **SDOS1** even though they are reachable? We found out that the main reason for this is that the current code-coverage-based feedback is not sufficient to steer the fuzzer efficiently towards more complex and interesting DY traces finding security vulnerabilities. To establish this, we analyzed the final corpora of the experiments that did not discover **SIG** or **SDOS1**. We first determined the code coverage of the corpus by executing all test cases in the corpus and storing the total code coverage. We tested and observed that applying any subset of the required mutations for finding **SIG** or **SDOS1** on traces from the *Seed Corpus* did not improve the code coverage. Therefore, such mutations, when applied one by one, will not be kept in the *Corpus* (as they are deemed uninteresting) because the *Corpus* already *exhausts* the coverage-based feedback entropy: the feedback metric is no longer able to measure progress. This happens when the *Corpus* becomes too large. It remains possible to find the required mutations before that point. Past that point, it remains possible (although very unlikely) to apply all required mutations at once. This is also why we recommend running multiple fuzzing campaigns versus only one for a longer period of time.

6.2.5. Execution Performance. We gathered the executions per second for the PUTs *OpenSSL* 1.1.1k, *LibreSSL* 3.3.3, and *wolfSSL* 5.4.0. For *wolfSSL*, we also included a sanitized binary (ASAN) [49]. Overall, our TLS fuzzer is fast and can reach > 770 executions per second on a single core with *LibreSSL* as PUT. *tlspuffin* fuzzing *OpenSSL* is about half as fast as with *LibreSSL* with > 320 executions per second. When fuzzing *wolfSSL*, *tlspuffin* reaches a similar performance with > 340 executions per second. With ASAN enabled, *wolfSSL* performance reduces by about 50% to > 160 executions per second, which is to be expected [49]. We observed that *tlspuffin* spends ca. 84% of CPU time in the PUT, and ca. 15% while mutating traces. Therefore, any future optimizations should be concerned with lowering PUT execution time or increasing the mutator's performance.

We claimed *tlspuffin* is delightfully parallel, and we indeed observed that executions per second of the PUT scales linearly with the amount of available cores. More detailed statistics can be found in Appendix D.5.

7. Conclusion and Future Work

In this paper, we propose the novel concept of DY fuzzing: we design a generic DY fuzzer, implement a DY fuzzer for TLS, and conduct a comprehensive evaluation thereof. The *tlspuffin* tool is a first step in deploying this approach but is already a full-fledged fuzzer that found new vulnerabilities in *wolfSSL*. More generally, this new approach connects fuzzing and Dolev-Yao style formal models and offers various directions for improvements, extensions, and applications.

Improve the DY fuzzer feedback. We established in Section 6.2.4 that the coverage-based feedback was subject to exhaustion and was not ideal to promote semantical diversity (in the sense of the DY model). It is not the fuzzing space that gets exhausted first, but the feedback space. There is also an interesting parallel with *overfitting* in machine learning. One could argue that *tlspuffin* currently overfits by trying to achieve the maximum code-coverage. Similarly to *dropout layers* in machine learning, a dropout in *tlspuffin* could randomly accept allegedly uninteresting test cases in order to *regularize* the fuzzing corpus.

More generally, future work should focus on finding a domain-specific feedback metric that takes advantage of our structured approach. For instance, a coverage metric over the DY attacker behaviors space (*e.g.* DY traces) could be defined and used. Ideally, the feedback metric would combine code-coverage with DY-related information: hitting the same code with (semantically) different adversarial behaviors should not be considered the same. Moreover, DY models can be reasoned about using state-of-the-art automated verifiers. DY fuzzers could rely on such verifiers to proxy closeness to attack traces, *i.e.* measure progress towards finding attacks. Such metrics could also be beneficial to our generative mutations that are currently random and blind.

Broaden the Scope. Currently, the *tlspuffin Objective Oracle* captures memory-safety bugs and authentication violations. However, classical DY verification of specifications operates on a richer class of properties. For instance, secrecy properties are expressed by the attacker's inability to *deduce* a term corresponding to allegedly confidential data. To reason about attacker deduction, we would need to *reinterpret* the bitstrings obtained by output actions as terms (computing the inverse of $[\![\cdot]\!]$) and then leverage decision procedures for deduction (*e.g.* [1]). Realizing this *reinterpretation* raises interesting research questions. Another interesting property is *functional correctness* with respect the an underlying protocol DY model. Finally, recent work allows verifying privacy properties [20], *e.g.* anonymity, that provide a challenging opportunity for extending this work.

Another direction is to augment the attacker capabilities by the additions of more mutations (*e.g.* modifying the channels and their configurations) and operators in Σ (*e.g.* systematic dummy values for each type). Note that the trade-off size of search space versus capabilities needs to be balanced.

Apply to more targets. We also want to apply DY fuzzing to more targets. Candidates for TLS PUT could be Microsoft's closed-source Schannel or the open-source Mbed TLS. We also plan to improve *sshpuffin* for SSH. DY fuzzers should be applied on other protocols too: *e.g.* the aforementioned WPA2 protocol and closed-sources or remote targets like those found in mobile telecommunication systems or industrial protocols, etc.

Acknowledgements

The *tlspuffin* tool was initially designed and implemented at LORIA (Inria, CNRS, Université de Lorraine), in Nancy, France. Extensions for *wolfSSL* and the SSH protocol was later added during a collaboration with Trail of Bits from New York, USA. This work has also been partly supported by the ANR Research and teaching chair in AI ASAP (ANR-20-CHIA-0024) and ANR JCJC project ProtoFuzz.

We also thank Alexander Knapp for his comments on an early version of this paper.

References

[1] M. Abadi and V. Cortier. "Deciding knowledge in security protocols under equational theories". In: *Theor. Comput. Sci.* 367.1-2 (2006).

[2] M. Abadi and C. Fournet. "Mobile values, new names, and secure communication". In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 2001.

[3] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P. Strub. "The Last Mile: High-Assurance and High-Speed Cryptographic Implementations". In: *Symposium on Security and Privacy (SP)*. IEEE, 2020.

[4] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, et al. "DROWN: Breaking TLS Using SSLv2". In: *USENIX Security Symposium*. 2016.

[5] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. "FUDGE: Fuzz Driver Generation at Scale". en. In: *Symposium on the Foundations of Software Engineering (ESE/FSE)*. ACM, 2019.

[6] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. "SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr". en. In: *Information Security*. Vol. 4176. Springer, 2006.

[7] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. "SoK: Computer-Aided Cryptography". In: *Symposium on Security and Privacy (SP)*. IEEE, 2021.

[8] D. A. Basin, R. Sasse, and J. Toro-Pozo. "Card Brand Mixup Attack: Bypassing the PIN in non-Visa Cards by Using Them for Visa Transactions". In: *30th USENIX Security Symposium*. USENIX Association, 2021.

[9] D. A. Basin, R. Sasse, and J. Toro-Pozo. "The EMV Standard: Break, Fix, Verify". In: *Symposium on Security and Privacy (SP)*. IEEE, 2021.

[10] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams. "The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform". In: *Communications of the ACM* 64.8 (2021).

[11] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *Symposium on Security and Privacy (SP)*. IEEE, 2015.

[12] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele. "DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code". In: *European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021.

[13] K. Bhargavan, B. Blanchet, and N. Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate". In: *Symposium on Security and Privacy (SP)*. IEEE, 2017.

[14] K. Bhargavan, V. Cheval, and C. Wood. "A Symbolic Analysis of Privacy for TLS 1.3 with Encrypted Client Hello". In: *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2022.

[15] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. "Implementing TLS with Verified Cryptographic Security". In: *Symposium on Security and Privacy (SP)*. IEEE, 2013.

[16] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub. "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS". In: *Symposium on Security and Privacy (SP)*. IEEE, 2014.

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

[18] H. Böck, J. Somorovsky, and C. Young. "Return Of Bleichenbacher's Oracle Threat (ROBOT)". en. In: *USENIX Security Symposium*. 2018.

[19] *boofuzz: Network Protocol Fuzzing for Humans*. https://boofuzz.readthedocs.io/.

[20] V. Cheval, S. Kremer, and I. Rakotonirina. "DEEPSEC: Deciding Equivalence Properties in Security Protocols Theory and Practice". In: *Symposium on Security and Privacy (SP)*. IEEE, 2018.

[21] V. Cortier and S. Kremer. "Formal Models and Techniques for Analyzing Security Protocols: A Tutorial".

In: *Foundations and Trends in Programming Languages* 1.3 (2014).
[22] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. "A Comprehensive Symbolic Analysis of TLS 1.3". en. In: *Conference on Computer and Communications Security (CCS)*. ACM, 2017.
[23] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe. "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication". In: *Symposium on Security and Privacy (SP)*. IEEE, 2016.
[24] *CVE - CVE-2009-3555*. https://www.cve.org/CVERecord?id=CVE-2009-3555.
[25] *CVE - CVE-2014-0160*. https://www.cve.org/CVERecord?id=CVE-2014-0160.
[26] *CVE - CVE-2014-1266*. https://www.cve.org/CVERecord?id=CVE-2014-1266.
[27] J. De Ruiter and E. Poll. "Protocol State Fuzzing of TLS Implementations". In: *USENIX Security Symposium*. 2015.
[28] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. "Implementing and Proving the TLS 1.3 Record Layer". In: *Symposium on Security and Privacy (SP)*. IEEE, 2017.
[29] A. Delignat-Lavaud, C. Fournet, B. Parno, J. Protzenko, T. Ramananandro, J. Bosamiya, J. Lallemand, I. Rakotonirina, and Y. Zhou. "A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer". In: *Symposium on Security and Privacy (SP)*. IEEE, 2021.
[30] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on information theory* 29.2 (1983).
[31] N. Drucker and S. Gueron. *Selfie: Reflections on TLS 1.3 with PSK*. ePrint IACR 347. 2019.
[32] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "AFL++ : Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, 2020.
[33] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Conference on Computer and Communications Security (CCS)*. ACM, 2022.
[34] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky. "Analysis of DTLS Implementations Using Protocol State Fuzzing". en. In: *USENIX Security Symposium*. 2020.
[35] *Fuzzowski Network Fuzzer*. https://github.com/nccgroup/fuzzowski.
[36] G. Girol, L. Hirschi, R. Sasse, D. Jackson, D. Basin, and C. Cremers. "A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols". en. In: *USENIX Security Symposium*. 2020.
[37] A. Hazimeh, A. Herrera, and M. Payer. "Magma: A Ground-Truth Fuzzing Benchmark". In: *Proc. ACM Meas. Anal. Comput. Syst.* 4.3 (2020).
[38] X. Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009).
[39] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *IEEE Transactions on Software Engineering (TSE)* (2019).
[40] S. Meier, B. Schmidt, C. J. F. Cremers, and D. Basin. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *International Conference on Computer Aided Verification (CAV)*. Vol. 8044. Springer, 2013.
[41] B. P. Miller, L. Fredriksen, and B. So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Communications of the ACM* 33.12 (1990).
[42] B. Möller, T. Duong, and K. Kotowicz. *This POODLE Bites: Exploiting The SSL 3.0 Fallback*. en. Tech. rep.
[43] K. G. Paterson and T. van der Merwe. "Reactive and Proactive Standardisation of TLS". en. In: *Security Standardisation Research*. International Publishing, 2016.
[44] V.-T. Pham, M. Böhme, and A. Roychoudhury. "AFLNET: A Greybox Fuzzer for Network Protocols". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020.
[45] E. Poll, J. D. Ruiter, and A. Schubert. "Protocol State Machines and Session Languages: Specification, Implementation, and Security Flaws". In: *Security and Privacy Workshops (SPW)*. IEEE, 2015.
[46] L. Project. *libFuzzer – a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html.
[47] A. T. Rasoamanana, O. Levillain, and H. Debar. "Towards a Systematic and Automatic Use of State Machine Inference to Uncover Security Flaws and Fingerprint TLS Stacks". In: *European Symposium on Researchcin Computer Security (ESORICS)*. Springer, 2022.
[48] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. 2018.
[49] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 Annual Technical Conference*. USENIX, 2012.
[50] J. Somorovsky. "Systematic Fuzzing and Testing of TLS Libraries". In: *Conference on Computer and Communications Security (CCS)*. ACM, 2016.
[51] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. "Dependent Types and Multi-Monadic Effects in F*". en. In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 2016.

[52] *tlspuffin Github repository*. https://github.com/tlspuffin/tlspuffin. A tarball with some artifacts discussed in the paper is also available here: https://drive.google.com/drive/folders/1ZOAkmMO-IviQ6FiPoWBJyCj_p1rjbny1?usp=sharing.

[53] M. Vanhoef and F. Piessens. "Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2". en. In: *Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[54] M. Vanhoef and F. Piessens. "Release the Kraken: New KRACKs in the 802.11 Standard". In: *Conference on Computer and Communications Security (CCS)*. ACM, 2018.

[55] G. Vranken. *Cryptofuzz project*. https://guidovranken.com/2019/05/14/differential-fuzzing-of-cryptographic-libraries/. 2020.

[56] M. Vučinić, G. Selander, J. P. Mattsson, and T. Watteyne. "Lightweight Authenticated Key Exchange With EDHOC". In: *Computer* 55.4 (2022).

[57] M. Zalewski. *American Fuzzy Lop - Whitepaper*. https://lcamtuf.coredump.cx/afl/technical_details.txt. 2016.

[58] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. "HACL*: A Verified Modern Cryptographic Library". In: *Conference on Computer and Communications Security (CCS)*. ACM, 2017.

Appendix A. List of Changes

- v1.0, January 18, 2023: initial version.

Appendix B. Related Work: Program Verification

We argue that verification methods targeting implementations of cryptographic protocols have currently two drawbacks. The first one is *scalability*. Using a proof oriented programming language such as *F** [51] requires a dedicated protocol implementations and significant effort. As an illustration, proofs for the record layer of QUIC [29] written in *F** required ca. 20 person months and do not cover the much more complex handshake protocol. For TLS 1.3 the proof is also limited to the record layer [28] and for TLS 1.2 the proof (using *F7*) does not cover the full handshake [15]. For particular cryptographic primitives, there have been efforts to provide end-to-end proofs of highly optimized implementation resisting side channel attacks, using the EasyCrypt proof assistant and the Jasmin domain-specific language and compiler [3]. No protocols were proven this way.

The second, related, short-coming is that none of these approaches applies to *existing, deployed* implementations such as our PUTs *OpenSSL*, *LibreSSL*, and *wolfSSL*. While some formally verified cryptographic libraries, such as *HACL** [58], written in *F**, start being deployed in production systems, such examples are still the exception, and cover the cryptographic library, not the protocol.

```
let rsa_certificate = term! {
  fn_certificate13(...)
};

let certificate = term! {
  fn_encrypt_handshake(
    (@certificate_rsa),
    (fn_sh_transcript(((server, 0)))),
    (fn_server_share(((server, 0)))),
    fn_seq_0,
    ...
  )
};
```

Figure 3: Example usage of the term DSL

The *DY** [12] verification framerwork scales to whole protocols but only operates on programs written in the *F** language, thus excluding existing code bases written in other languages (C, Java, C++, Rust, Go, etc.) such as our PUTs.

Finally, program verification of existing code, for instance in C, has been made possible with tools like *Frama-C* [10] combined with security proofs in tools like *Easycrypt* [7] and secure compilers like *CompCert* [38]. However, this approach does not currently scale beyond simple cryptographic primitives, excluding protocol implementations.

Appendix C. DY Fuzzing

C.1. DSL for DY Traces

When defining traces for the *Seed Corpus*, the users of DY fuzzers are supposed to manually craft traces. The *puffin* and *tlspuffin* tool simplify this task by providing a DSL which allows declaring traces. In Figure 3, we provide examples of terms which can be used to build a whole trace. The DSL is implemented through Rust's `macro_rules!` feature. The example creates an encrypted `Certificate` message. It references a sub-term in line 7, references knowledge through a query in lines 8–9 and uses a constant in line 10. The Rust variable `certificate` can be used to create a trace. We provide complete traces in this DSL for defining the whole *Seed Corpus* in [52].

C.2. Notes on Mutations

C.2.1. Mutation Redundancy. Note that the result of the *Swap* mutation between two terms whose one is a subterm of the other can also be achieved through a *Replace-Reuse* mutation. There exist other such redundancies. Redundancies in mutators is actually a common pattern, *e.g.* it also occurs in the *havoc mutator* of bit-level fuzzers like AFL++ [32].

C.2.2. Mutation Failures. The application of a mutation is a random process that can fail at two levels: (i) when executed later using the *Harness* or (ii) when computing

the mutated trace. An example of failure (i) is the *Replace-Match* mutation replacing $f(t_1, \ldots, t_k)$ with $f'(t_1, \ldots, t_k)$ such that $f(t_1, \ldots, t_k) \in \mathsf{dom}(\llbracket\cdot\rrbracket_\phi)$ (the computation succeeds) but $f'(t_1, \ldots, t_k) \notin \mathsf{dom}(\llbracket\cdot\rrbracket_\phi)$ (the computation fails). Those failures are to be expected and are dealt with by the fuzzing loop; the mutated trace will be deemed uninteresting and dropped. Failures (ii) can happen when the chosen mutation cannot be applied while respecting the aforementioned mutation constraints, notably typing constraints. In those cases, the mutation is simply skipped.

Appendix D. Implementation: The *tlspuffin* Fuzzer and its Evaluation

D.1. Determinism

Deterministic behavior is very important during automated software testing, because results should be reproducible. This can be achieved by seeding Pseudorandom Number Generator (PRNG) with static values. Therefore, this PRNG should be used to generate randomness when having to resolve random choices for the presented mutations. This way, it is possible to replay mutations by providing the same seed twice and write unit tests. It is noteworthy, that each execution of a trace yields an identical run, under the assumption that the PUT is deterministic. We explain in Section 5, how we managed to make the PUTs behave deterministically. Note that, even if the PUT is not deterministic, its non-deterministic behaviors are likely to solely concern the content of some random fields such as random nonces or ephemeral keys, whose variations are very unlikely going to impact how DY traces are executed, at the logical level.

D.2. Mutation Benchmark

We provide additional data about the mutation evaluation from Section 6.2.3. Instead of only presenting summed values, we go here over the mutation specific numbers for the **SIG** and **SDOS1** vulnerabilities.

For **SIG**, we found that the *Replace-Match* mutation must be executed on average 336 times before the desired mutation is performed, *i.e.* when it replaces the certificate in the `Certificate` message. Then the same mutation must be executed again on average 412 times to set an invalid signature algorithm in the `CertificateVerify` message. Lastly, the *Generate* mutation that sets an invalid signature algorithm must be applied on average 61 times to change the signature in the same message.

For testing the effect of additional type-constraints, we added types for certificates and private keys instead of using generic byte arrays. This reduced the required tries by 195 for *Replace-Match* and 69 for the second mutation step. The required mutation tries for the *Generate* stayed the same, which is to be expected because it does not depend on certificates of private keys. (Note that we did not include those additional constraints in the main version of *tlspuffin* as they slightly reduce the attacker capabilities and were not strictly necessary.)

For **SDOS1**, we need at least five mutations in a specific order: *Repeat*: 9, *Replace-Reuse*: 880, *Replace-Match*: 1967, *Remove-and-Lift*: 24, and *Replace-Reuse*: 173.

D.3. Notes on *tlspuffin* as an Analysis Framework

As for *TLS-Attacker* [50], *tlspuffin* also allows defining specific protocol flows for TLS libraries using our handy DSL and executing them over TCP. This way, users can test for the absence of known vulnerabilities, extract fingerprinting information, or send arbitrary custom TLS messages. Thus, despite not being its main goal, *tlspuffin* also offers this *TLS-Attacker* use case.

D.4. Notes on Triaging BUF

As explained in Section 5.3, our implementation of a DY fuzzer offer various features that ease bugs triaging and obtaining deep understandings of their root causes. We now explain the methodology we followed to do so, taking as an illustration the **BUF** vulnerability *tlspuffin* discovered (see the full vulnerability report in [52]).

1) Confirm the bug against up-do-date PUT freshly cloned and built using the TCP feature: We execute the on-disk test case from the *Objective Corpus* against a standalone *wolfSSL* TCP server which has the same version as the PUT used during fuzzing. We observed the stack buffer overflow using ASAN.
2) Plot the trace to understand the flow of messages that triggers the bug. It shows the attack DY trace with all the attacker terms in a tree structure. The one for **BUF** (see [52]) indicates that two maliciously formed `ClientHello` messages, which contain multiple duplicate cipher suites, are responsible for the bug (mentioned Section 6.1.1 and detailed in Remark 3 below).
3) Dynamically analyze the bug: Using the `gdb`/`ldb` debugger on the whole *tlspuffin* binary and executing the attack DY trace step-by-step on the PUT, we can pinpoint the exact conditions required to trigger the bug. In particular for **BUF**, we obtained this way a full understanding of the attack and its root causes, as shown next.

Remark 3 (**BUF** root causes description)**.** *Debugging tlspuffin and wolfSSL when executing the **BUF** attack DY trace step-by-step, we observed two things:*

a) When receiving a `ClientHello` message, the server computes the resulting negotiated list of ciphers based on the ciphers the server offers, stored locally in ssl−>suites, and the ciphers offered by the client suitesC, stored in the received `ClientHello` message. However, the negotiated list is computed by a multi-set intersection between ssl−>suites and suitesC instead of a set intersection. When suitesC contains duplicates (say k duplicates) of a cipher suite that occurs (say just once) in

ssl−>suites, then the resulting list will contain all those duplicates as well.

b) *After the first full, happy-flow of the initial full handshake, because the first maliciously crafted* `ClientHello` *does not have a* `supported_groups` *extension but tries to resume a previous session, the server rejects it and answers back a* `HelloRetryRequest`*. However, before rejecting it, it still computes the resulting negotiated list and stores it in ssl−>suites without reverting this write when aborting and sending back the* `HelloRetryRequest` *message. Therefore, when the second maliciously crafted* `ClientHello` *is received, the server will compute the negotiated list again using the modified ssl−>suites, which contains k duplicates, and suitesC, which also contains k duplicates. which yields a list which is as long as k^2. Note that some bound checks avoid the list suitesC and the initial list ssl−>suites to be larger than 150. Because $k^2 > 150$ when $k \geq 13$, our attack bypasses those bound checks and triggers the stack buffer overflow.*

To summarize, the bug is due to a combination of flaws: a flaw in the negotiation computation routine, which did not expect duplicates, and a flaw in the `HelloRetryRequest` *handling which does not revert side effects. The length can be controlled by the attacker with k and can reach up to 44700 bytes. However, the bytes that can be written on the stack must be valid cipher suite encoding offered by the server, which limits the alphabet of available writable bytes.*

We stress that, for this bug to be reached, multiple preconditions need to be met: The TLS session must be resumed, a `HelloRetryRequest` *must be triggered with a malicious omission of* `supported_groups`*, and two cipher suite lists with duplicates need to be sent. Using our DSL for DY traces, we confirmed that all of those are required, i.e. removing any of those prevents the bug to be triggered. Such complex, logical adversarial behaviors are reached in a matter of a few mutations with DY fuzzers like tlspuffin.*

Continuing the overall methodology:

4) Sometimes it was necessary to test-out hypotheses by constructing minimized trace variants using our DSL. When the variant still triggers the bug, we could go back to (3). Sometimes, we re-ran a fuzzing campaign with the variant as seed and let the fuzzer find variants that trigger the bug and go back to (3). We did so for **BUF**, and *tlspuffin* found a smaller trace, where the first full handshake is completely skipped. Instead, *tlspuffin* found that the first malformed `ClientHello` message can pretend to resume a non-existent session. From this trace written in our DSL, we could trial and error removing parts of the trace until reaching a minimal PoC. Finally, we could use the bitstring extractions to produce a `netcat` command as PoC.

D.5. Detailed Performance Results and Plots

See Table 3 and enlarged version of Figure 2 in Figure 4.

PUT	Version	ASAN	Executions/s per core	Executions/s
OpenSSL	1.1.1k	✗	320	4770
LibreSSL	3.3.3	✗	770	9920
wolfSSL	5.4.0	✓	340	4330
wolfSSL	5.4.0	✗	160	2040

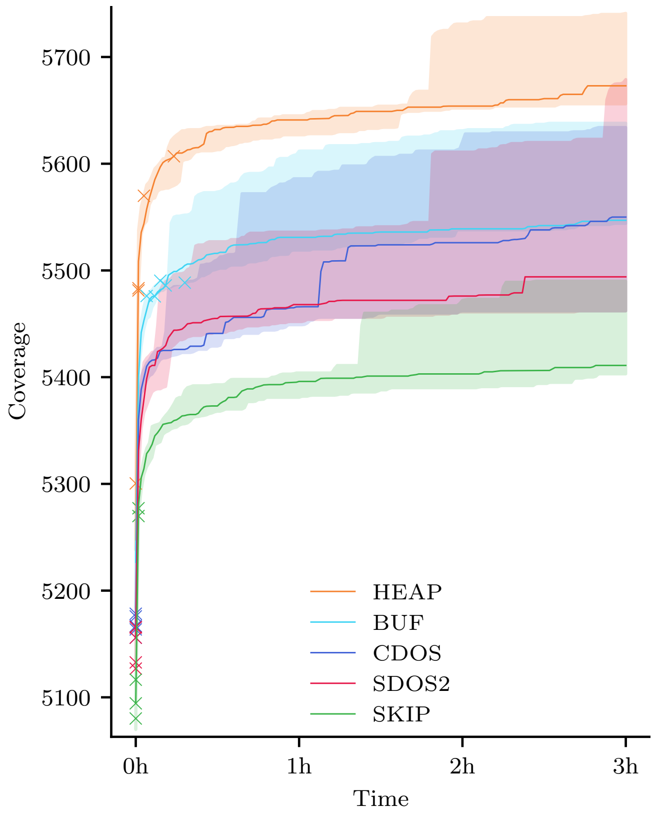
TABLE 3: Performance of selected *tlspuffin* builds. Executions/s per core might differ from executions/s when multiplied by the total number of cores (12), as fuzzing might run with slightly different speed on different cores.

D.6. Responsible Disclosure

Using *tlspuffin*, we discovered four new vulnerabilities. Each of them has been responsibly disclosed to and fixed by *wolfSSL*. They have also been filed as CVEs as indicated in Table 2. For each of them, we offered our help, notably proposing fixes (some of them have been tested by *tlspuffin*). Over the course of about 3 months *wolfSSL* fixed all vulnerabilities with a mean time of 6 days until the fix landed on the main branch and a mean time until fixes were released of 26 days. A detailed timeline can be found below. Users of *wolfSSL* like the OpenWRT[8] were informed by the publication of the CVE on *wolfSSL* release days.

08/12/2022 Contacted *wolfSSL* to set up a secure channel.
08/12/2022 Reported CVE-2022-38153 and CVE-2022-38152.
08/12/2022 Confirmed and fixed CVE-2022-38152.
08/16/2022 Confirmed CVE-2022-38153.
08/17/2022 Fix CVE-2022-38153.
08/30/2022 Release of a fix version 5.5.0.
09/12/2022 Reported CVE-2022-39173.
09/12/2022 Confirmed and fixed CVE-2022-39173.
09/28/2022 Release of a fix version 5.5.1.
10/09/2022 Reported CVE-2022-42905.
10/10/2022 Confirmed and fixed CVE-2022-42905.
10/28/2022 Release of a fix version 5.5.2.

8. https://openwrt.org/advisory/2022-10-04-1



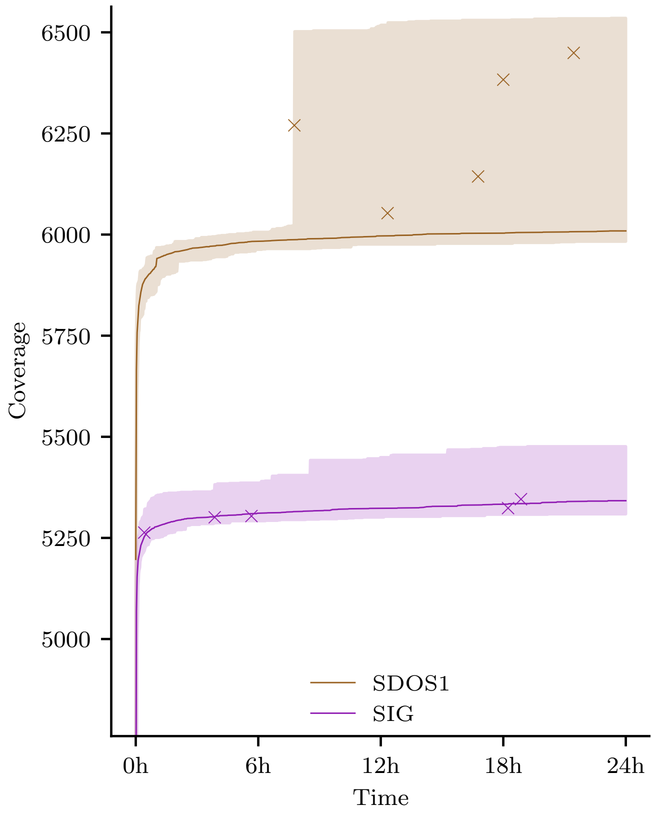


Figure 4: Larger version of Figure 2.