DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic 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

v1.2† — December 1, 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, *e.g.* attacks that exploit flawed protocol logic. Automated formal verification methods, based on the Dolev-Yao (DY) attacker, formally define and 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 such implementations are secure. Unfortunately, this blind spot hides numerous attacks, such as 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 novel 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 represented as formal terms (*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 extremely hard to get right. Critical and widely used cryptographic protocols such as Transport Layer Security (TLS) have been repeatedly found to be flawed, both in their design and their implementation, usually with dramatic consequences. For the last decades, security researchers have identified different relevant classes of attacks and methods to prevent them.

*. 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.

Logical attacks and formal verification. Since the 1980s [36], the formal methods community has identified and mathematically defined the ***Dolev-Yao (DY) attacker*** and the corresponding class of ***logical attacks*** [10]. The DY attacker is an active network attacker who treats exchanged messages as formal terms (in a *term algebra*): this representation reveals the messages' internal algebraic structure but not their underlying concrete values as bitstrings. This attacker can intercept, eavesdrop, modify and synthesize messages (viewed as terms) in between honest agents, and also actively participate in protocol sessions. In particular, they can use cryptographic primitives (*e.g.* encrypt, sign, decrypt) by applying function symbols (also called *operators*) that encode the algebraic properties of those primitives (*e.g.* decrypting a ciphertext $\mathsf{senc}(m, k)$ with the right key yields the plaintext: $\mathsf{sdec}(\mathsf{senc}(m, k), k) = m$). Logical attacks are the attacks captured by this DY attacker, and from now on, we refer to them as ***DY attacks***. Typical examples are attacks that exploit flaws in the *protocol logic* such as Machine-in-the-Middle (MITM), authentication bypass, downgrade, replay, impersonation attacks.

Much work has focused on developing formal verification methods and tools aimed at finding or proving the absence of DY attacks at the design level –that is, in *cryptographic protocol specifications*. It has persistently been motivated by the prevalence of such ***design-level DY attacks*** and the high complexity of manually finding them. For instance, TLS 1.2 and early drafts of TLS 1.3 specifications suffered from serious DY attacks such as complete authentication bypasses [28, 16, 26, 27, 18, 58, 19, 7, 37]. Similar attacks impacted many other protocols such as WPA2 [74, 75], credit card payment systems [12, 11], etc. 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 [62, 77]).

However, specifications are simply an abstraction of the program that end-users deploy and run, programs that are themselves plagued with frequent implementation bugs. These may introduce additional vulnerabilities, notably ***implementation-level DY attacks***. Examples thereof are abundant and well illustrated by the extensive history of such attacks on TLS: [30, 14, 33, 40, 28, 58, 21, 65]. More generally, even a formally proven specification can result in a flawed and insecure implementation. Sadly, fundamental problems in program verification are still unsolved and coping with (combinations of) features of real-world systems such as pointer aliasing or complex control-flow is still out of scope, precluding verification of 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 but not the implementations thereof (*cf.* Section 2.3).

Memory-related vulnerabilities and fuzzing. In this paper, we are concerned with finding *implementation-level DY attacks* in large cryptographic protocol code bases. For this, we build on ***fuzz testing***, which has been developed since the 1990s [57] and is now the gold standard for testing security software. It has become a key part of software development practices; *e.g.* Google [8], 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). If some *feedback* metric (*e.g.* code-coverage) deems the mutated test case interesting, it is added to the *Corpus*. When executing test cases, an *objective oracle* observes whether a *security policy* violation occurs (*e.g.* memory corruption).

However, state-of-the-art fuzzers are unable to capture the class of ***implementation-level DY attacks*** for two main reasons. First, they fail to effectively capture the DY attacker, in particular the ability of structural modifications on the term representation of messages (*e.g.* re-signing a message with some adversarial-controlled key) which are a prerequisite to capture DY attacks. We emphasize that DY attacks may trigger protocol or memory vulnerabilities. Second, they are unable to detect a security violation at the protocol level; *e.g.* for the attacks that trigger protocol vulnerabilities, which are not memory-related, such as an authentication bypass. (We discuss this gap further in Section 2.3.)

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

DY Fuzzing: a new fuzzing technique. 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 DY attacks in protocol implementations. This requires a re-design of most of the fuzzer components.

The **test cases** become formal and structured *DY traces*. Those represent executions of protocols against a DY attacker in the (formal) DY model: networking actions (inputs, outputs) and adversarial manipulations over exchanged messages that are abstracted away by terms and idealized cryptography [10]. The DY fuzzer is thus capable of intercepting, eavesdropping, modifying, and synthesizing messages. We design new domain-specific **mutations** over DY traces for exploring the search space made of all the DY traces. 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.

The *messages as terms* paradigm guides DY attackers and thus our DY fuzzer in how they interact with messages and cryptography. It is on purpose that our fuzzer inherits this standard DY assumption: it narrows down the search space to DY attacks and exclude some other attacks, *e.g.* when bit-level transformations are required and cannot be described at the term-level. The DY attacker and attacks are motivated by and grounded on decades of research and practical attack finding [10].

Next, we define domain-specific **fuzzing security policies and objective oracles**. Indeed, the usual policies and objective oracle for spatial and temporal memory errors (*e.g.* AddressSanitizer (ASAN) [68]) are unable to detect *DY attacks triggering protocol vulnerabilities*. We still enable them, as they are useful to find *DY attacks triggering memory vulnerabilities*, *e.g.* memory bugs that can only be triggered from specific states that are hardly reachable using standard fuzzing. As we shall see, DY traces are very good at exploring such *deep states*. To be able to detect protocol vulnerabilities, we additionally consider formal DY properties, *e.g. strong agreement*, that express the absence of protocol vulnerabilities. Those properties are translated into security policies and objective oracles. This way, any violation thereof reached through fuzzing is detected and flagged as an attack candidate. We stress that a DY fuzzer does not require a complete protocol DY model, but only a DY attacker model (by 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 modules that are of independent interest. First, the protocol- and PUT-agnostic DY fuzzer module *puffin* is built on top of the fuzzing library *LibAFL* [39], where we re-implemented most of the fuzzing components. 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 [73] is ca. 16k *Rust* LoC. Extending the fuzzer to new PUT requires little work (hundreds of *Rust* LoC) but extension to new protocols is more costly (thousands of *Rust* LoC) as it requires protocol-dependent concretization, *i.e.* computing bitstrings out of terms.

Our fuzzer is fast (>700 executions/second/CPU core) and allows parallel processing. We let *tlspuffin* run on the aforementioned TLS PUTs, which found seven vulnerabilities, including four new CVEs on *wolfSSL* (among which one critical and two high). We also ran three state-of-the-art stateful protocol fuzzers and one test-suite and confirm that none was able to find any of the seven vulnerabilities. We explain why they were out of their scope due to inherent limitations of previous approaches. We perform benchmarks comparing our fuzzers to other stateful protocol fuzzers. *tlspuffin* achieves a similar amount of code coverage but the covered code is often incomparable. However, our analysis shows that code coverage is an unsatisfactory metric to evaluate the DY fuzzer's ability to find attacks (which was already noted for standard, bit-level fuzzers [50]).

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 a DY attacker and the class of DY attacks. We propose a new and complete system design.
2) We propose *tlspuffin*: a full-fledged, modular, and efficient implementation in *Rust* of this fuzzer design.
3) We evaluate and compare *tlspuffin* on several TLS 1.2 and TLS 1.3 libraries and (re)found seven vulnerabilities not found by others, including four new ones (one critical, two high, and one medium).

Outline. We first recall some background about TLS and fuzzing and discuss related work (Section 2). Then, we provide a DY model that allows us to define the fuzzer's search space (Section 3). We made this model generic enough in order to then directly link it to the executions of an implementation. 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 qualitatively and quantitatively evaluate and compare our tool with state-of-the-art protocol fuzzers (Section 6). Finally, we conclude by discussing directions for future work (Section 7).

2. Background

2.1. Case Study: The TLS Protocol

TLS [66] 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 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 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 endpoints, 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 [66, 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) 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 of the entire handshake with the certified key. (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.

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* [38, 80, 53, 55]. 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

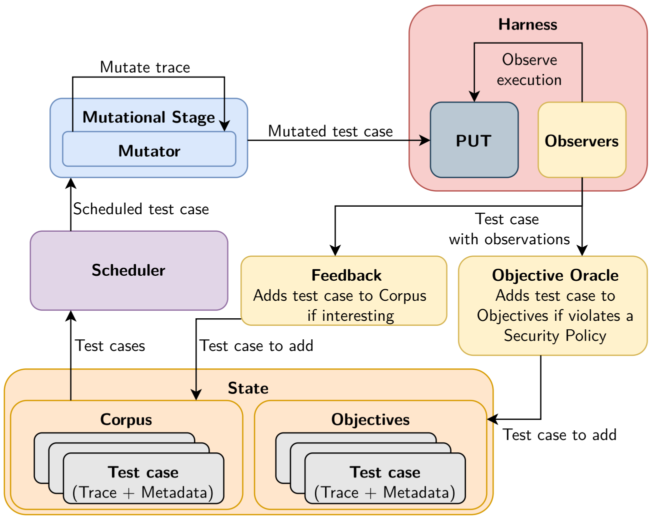


Figure 1: Fuzzer Architecture

executing the test cases to detect security violations in the PUT, such as a buffer overflow.

We are mostly using the terminology of *LibAFL* [39], 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

2.3.1. 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 [63, 76, 22, 42]. Such testing methodologies are adequate to find safety vulnerabilities with potential security implications (*e.g.* HeartBleed [30] and CloudBleed [29]), but are unable to capture DY attacks for two fundamental and intrinsic reasons given next, which we will revisit and exemplify in Section 6.1.5. (We provide a more thorough qualitative and quantitative comparison with state-of-the-art protocol fuzzers in Section 6.)

(1) Reachability Problem: DY attack states (triggering memory or protocol vulnerabilities) are **not reached** for two main reasons. *(1a) Existing harnesses* of many state-of-the-art bit-level fuzzers [38, 80, 53, 23, 79, 6] only send one flight of messages to the PUT and do not address the PUT statefulness.[1] This excludes vulnerabilities that require at least one round trip where the fuzzer needs to produce test-cases made of multiple messages successively fed to the PUT that may depend on outputs the PUT produces as response of previous messages. *(1b) Mutation.* Even for bit-level fuzzers made amenable to protocols, it is overwhelmingly unlikely that bit-level transformations (*e.g.* random bit flips and basic arithmetic operations) on the network packets express valid and logical transformations over messages such as uses of cryptography to perform *e.g.* impersonation, downgrade attacks, relay attacks, etc. Indeed, the latter often require numerous, structured, and synchronized modifications in the packets. *E.g.* almost all TLS 1.3 messages are encrypted and integrity-protected[2], which drastically reduce the probability of finding meaningful bit-level mutations (upper-bounded by the negligible probability to break the cryptographic suites). On the contrary, the message as formal term paradigm allows DY fuzzers to produce meaningful message mutations. This *Mutation* limitation affects all fuzzers based on bit-level mutations such as [38, 80, 53, 63, 76, 22, 44, 42, 51, 46, 54, 70, 49, 63, 60, 71] or/and based on mutations that can only modify a hard-coded selection of message fields [51, 54, 70, 49, 71]. We stress that 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) Detection Problem. Protocol vulnerabilities are **not detected**: the classical spatial and temporal memory errors-related security policies and objective oracle are unable to detect when a protocol vulnerability 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. For the first time, DY fuzzers can detect DY attacks triggering protocol vulnerabilities.

2.3.2. Model-based protocols fuzzing. Some prior work extended the *stateful fuzzing approach* [9] and use input-output protocol Finite State Machines (FSM) as a behavioral abstraction of the PUT and use *differential fuzzing* to detect potential bugs [49, 48] or manually inspect the inferred FSM [65, 33, 14, 40, 64]. They are tailored to capture bypass authentication attacks or other violations of the intended state machine flows. However they are unable to capture the entire class of DY attacks since they cannot tamper with the message contents (except for potentially a finite number of selected/hard-coded values [71, 49]) and thus suffer from the limitation (1b); the same applies to many other testing methodologies and fuzzers [47, 67, 32, 48, 59].

Moreover, since the FSM is not specifically designed for security, the security policy violations detected by FSM-based techniques are not necessarily security attacks and require manual inspection, those techniques thus also inherit the *Detection Problem (2)*.

2.3.3. Program verification and secure compilation. We argue in Appendix B.1 that verification methods targeting implementations of cryptographic protocols (*e.g.* [72, 35, 34, 18, 4, 81, 15, 13, 10, 52, 5]), such as F^*, DY^*, *Jasmin*, etc. have currently two drawbacks that make them unsuitable to preclude implementation-level DY attacks in *existing* real-world protocol implementations: *scalability* to whole

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

2. Disabling cryptography is not satisfactory as it would only impact the record layer and would hide the attacks related to the use of cryptography.

large protocols and, more importantly, ability to *operate on existing, deployed* implementations.

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 [36] that is today the basis for numerous verification techniques [10] and real-world security analysis [16, 26, 27, 12, 43, 17] of protocol designs (but not of implementations). We now recall some preliminary basic definitions in this model and refer the curious reader to [25, 20].

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 an atom $\mathsf{true} \in \Sigma$.*

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 the DY 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* $\mathsf{checksign}$*, 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 [25, 20]. 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 4.1).*

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 *DY attacks*, which are formally defined as the class of attacks that can be triggered by executing a *DY trace* (defined below). The *DY attacker* is an attacker who can perform DY attacks: its behavior is defined as the set of all DY traces.

Intuitively, *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. We emphasize that x is not the expected output but a variable that points to the actual message that has been output. 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*[3].

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

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

If $t = x$, the attacker simply forwards the server's response to the client. But the attacker can adopt many attack strategies. We give a few examples. E.g., if $t = \mathsf{someError}$, 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 atom 0.*

Semantics: Definition. We now present 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 [25, 20]) 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:

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

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:

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

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 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* (starting with an empty initial attacker's state):

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

Remark 2. *As is standard in DY protocol verification, honest agents are usually specified in a formal language such as the applied π-calculus [20] (or multiset rewriting rules [56]). In that case, states in $\mathcal{S}$ correspond to π-calculus 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. We shall also instantiate agents' local states into PUT-specific session handlers (e.g.* `SSL*` *pointers in OpenSSL).*

Remark 3. *We note that while the interpretation $[\![\cdot]\!]$ transforms terms ($\mathcal{T}$) into messages (M), we do not need to assume the converse can be done, i.e. parsing messages as terms. When messages are bitstrings, it may actually not be possible to parse message protected by cryptographic primitives. E.g. when h is a hash function one cannot parse $[\![h(t)]\!]_\phi$ if $[\![t]\!]_\phi$ is not already known. This is why we use a variable x to refer to an output message, on which function symbols can nonetheless and* w.l.o.g. *then be applied by the attacker (through an attacker term t).*

3.3. DY Security Properties

We use the notion of *claims* to express security properties. Throughout a trace execution, one can record, in local states, claims that log in which states the honest agents are and what they believe to be their environment (*e.g.* their peer's identity). *Claims* are expressions $\mathsf{c}(m_1, \ldots m_i)$ where c is a symbol with an arity i and $m_1, \ldots m_i \in \mathsf{M}$ 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 will typically be created for clients and servers finishing handshake sessions while running claims are created as soon as m (*e.g.* session identifier) is available to them. 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 created so far for 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 i^{th} 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 [56].

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 DY 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* is used to implement $[\![\cdot]\!]$, that is to transform terms into protocol messages as bitstrings that can be sent to the PUT. One way to implement the mapper is to use the PUT's implementation of a primitive f to compute the concretization $[\![\mathsf{f}]\!]$. However, a different implementation (*e.g.* a reference or a custom implementation) could also be used, we shall call it *SignatureLib*. 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. This signature is not exhaustive but contains the symbols necessary to produce terms corresponding to all types of messages of TLS 1.2 and 1.3, making it possible to run full TLS sessions. However we currently only support a rather small number of cipher suites. 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 an atom 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, 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 this 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 *session handler* 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 shall instantiate the input and output functions as follows. 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. Next, $(s'_c, m) := \mathsf{output}(s_c)$ 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 E.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 the attacker's state. For instance, when a PUT server returns a `Server Finished`, it is possible to infer that the server believes it has finished the handshake.

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*[4] for all possible expected protocol configurations. When evaluating our fuzzer we solely use happy flows and no attack traces as seeds.[5]

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 of item *(iii)*, followed by a resumption handshake with the same agents. This yields 11 seeds in total for TLS (1.2 and 1.3).

4. 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.

5. Otherwise, attacks would always be found immediately and bias the evaluation. For real fuzzing campaigns, including attack traces on previous versions of the PUT, or stemming from a different implementation, is however desirable as it allows for regression testing and may also ease finding variants of an attack that was not properly fixed.

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

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.). We describe all mutations we consider for *tlspuffin* in Table 1 (split into action- and term-level mutations). We consider these to be a good basis for any DY fuzzer as they fully capture the DY attacker and are completely protocol (and obviously PUT) independent. See an ablation study and why each single mutation is useful in Section C.1.5.

Action-level mutations. The *Skip* mutation removes a random action from a trace. The *Repeat* mutation repeats 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 [14] and **SKIP** from Table 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 possibly using cryptographic primitives (*i.e.* for expressing a certificate swapping). These mutators require a description of test cases that specifies the structure of messages and the available cryptographic primitives, 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 atoms (*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* $A = \mathsf{in}(s, ch).\mathsf{out}(s, x)$ *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 triggers a HelloRetryRequest message due to the missing extension e_n, i.e. when removing the supported_groups extension. Then applying the Repeat mutation, we obtain the trace $A' =$ $\mathsf{in}(s, ch_k).\mathsf{in}(s, ch_k).\mathsf{out}(s, x)$. *We shall see in Section 6 that additional mutations, notably involving cryptographic operators, lead to a buffer overflow on wolfSSL when* $13 \leq k \leq 150$.

Mutation Constraints. Mutations can be applied only when specific constraints are fulfilled. In particular, we shall exclude mutations that yield traces that cannot be gracefully executed (through $[\![\cdot]\!]$) for example because of an ill-formed attacker term. We thus restrict to well-typed attacker terms according to the type system of *SignatureLib*. We also impose sane limits on the number of actions and attacker term sizes. We provide more details in Section D.2.1.

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 [68]. DY fuzzers *also* use those but we strive to go beyond since, as is, the fuzzer would silently miss protocol vulnerabilities.

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 mutual 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 claims : $\mathcal{S} \rightarrow \mathcal{P}(\mathsf{C})$. 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 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 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 achieved code coverage, the fuzzer decides whether A' is worth being stored in the *Corpus*. The fuzzer then proceeds to the next loop iteration.

5. Implementation: The *tlspuffin* Fuzzer

We present *puffin* and its derivatives, which altogether is a public FLOSS framework [73] 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 also using *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*.

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* [39] fuzzer library in order to implement an *evolutionary* fuzzing loop. We implement custom *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 declaratively defining DY traces (see Appendix D.1). We use the standard AFL-like code edge coverage map [39] (*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*.

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 ease bug triaging (see Section 5.2).

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 189 of its operators (see Section 4.1). *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. The *Harness* for *OpenSSL*/*LibreSSL* and for *wolfSSL* respectively required ca. 500 and 577 *Rust* LoC.

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 and Features

We review some challenges we tackled in building *puffin* and *tlspuffin*, which are detailed further in Appendix E.

Performance. Performance was key for our implementation choices. 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 chose *LibAFL* for its high performance and parallel processing. Different fuzzer clients can be running on different CPU cores and share the same *Corpus*. For this, we had to make many components serializable: the term algebra and its link with $[\![\cdot]\!]$ (how to concretize), DY traces, claims, etc. See Section C.1.4 for a performance evaluation.

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 attacker terms using functions to *e.g.* access fields in m (see projections from Example 2, as well as Remark 3). This way, he can treat m as a term. However, this yields a lot of redundant sub-terms to access the same fields. To simplify attacker terms, we decided that *puffin* would partially interpret m by extracting all sub-messages that can be accessed in plaintext and assigning them variables x_i, which are added to the attacker's state ϕ. Similarly, we observed that the fuzzer often had to build some complex but public terms in traces, which correspond to hashes of the current transcript. Those dramatically increase the size and complexity of traces. We implemented a feature that, *w.l.o.g.* allow the attacker to invoke a routine to compute such sub-messages without having to provide a full attacker term for it. We stress that those two modifications do not change the executions and the attacker's behaviors that can be explored.

Queries. We noticed that the way the attacker refers to its state ϕ was often not robust enough through successive mutations. *E.g.* consider a variable x and an attacker term $t = \mathsf{sdec}(x, k)$ in a trace $T = T_1.\mathsf{out}(c, x).\mathsf{in}(c', t)$. 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. 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 and its internal *Rust* type. A query is simply a conjunction of conditions over metadata (c, message kind, message type) to access the first matching message in the attacker's state ϕ. It also eases the writing of the seed traces.

Additional Features. It is possible to display (as trees) and execute traces which are stored on-disk against any PUT, even against arbitrary TCP clients or servers (including remote and closed-source). We are already using our tools to test for regressions in the supported PUTs, treating existing attack traces as regression tests. *tlspuffin* also offer features, notably its DSL, to allow developers to analyze TLS libraries with a similar goal of *TLS-Attacker* [41].

6. Results and Comparison with State-of-the-art Protocol Fuzzers

We now present our evaluation methodology and the obtained results qualitatively and quantitatively showing the peculiarities of DY fuzzing and where it shines, notably its superiority at finding DY attacks. All our experiments are reproducible with scripts available at [73].

6.1. Finding DY Attacks

Our first goal is to answer the following **Research Question:** *How does tlspuffin perform and compare with others at finding DY attacks?*

6.1.1. Methodology.

Vulnerabilities benchmark suite. To evaluate our work, we first establish a benchmark suite. The Magma project [45] proposes a consolidated benchmark suite of 20 memory-safety related bugs in *OpenSSL*, but none of these is a DY attack on which we could meaningfully evaluate DY fuzzing. A concrete example of such an out-of-scope vulnerability is the memory corruption in the ASN.1 encoder [31] which relies on a representation of zero as a negative integer. Such representation issues are out of scope of term-level modifications. Therefore, we selected 3 recent, known DY attacks (the 3 first vulnerabilities in Table 2) on *OpenSSL* and *wolfSSL*. To this initial ground-truth seed of known bugs, we add 4 newly discovered DY attacks. The result is a first relevant benchmark suite of DY attacks, that we plan to augment in the future.

Comparison with state-of-the-art fuzzers. According to the following criteria we selected state-of-the-art fuzzers to be compared with *tlspuffin*: support for stateful PUT, tested on at least one cryptographic protocol, active project in the last 5 years. These criteria resulted in the selection of AFLNet [63], StateAFL [60], and AFLnwe [3], which were already plugged into the ProFuzzBench [61] benchmark tool.[6] All our experiments were conducted on a server with 500 GB of RAM and 2 `AMD EPYC 7F521` processors with 16 physical cores each (but there are no strict minimum requirements for running *tlspuffin*).

6.1.2. *tlspuffin* performance. *tlspuffin* is fast and can reach > 770 execs/s on a single core with *LibreSSL* as PUT. Fuzzing *OpenSSL* or *wolfSSL* is about half as fast. Enabling ASAN reduces performance by about 50%, which is to be expected [68]. About 84% of CPU time is spent in the PUT, and ca. 15% in trace mutations. *tlspuffin* scales and parallelizes well, as we observed that execs/s scales linearly with the number of available cores. The compared fuzzers reach <10 execs/s due to the overhead of creating a subprocess for each execution and sending messages through TCP.

6.1.3. Fuzzing Result Comparison. We ran each of the aforementioned fuzzers (with ASAN) three times for 24h fuzzing campaigns on WolfSSL 5.3.0, which was affected by 6 vulnerabilities *tlspuffin* found; none of those 9 campaigns found any of them[7].

Moreover, *OpenSSL* and *wolfSSL* are well-tested software and fuzzers are already continuously testing them. In particular, Google's large-scale oss-fuzz project [69] continuously runs AFL++, honggfuzz, and libfuzzer [38, 53, 44] on *OpenSSL* as well as honggfuzz and libfuzzer on *wolfSSL*. The *wolfSSL* project itself runs 7 fuzzers internally every night (including a network fuzzer, libfuzzer, tlsfuzzer, and AFL) [78]. Those intensive fuzzing efforts were unable to find any of the 7 vulnerabilities from our benchmark suite.

In contrast, *tlspuffin* was able to find all 7 vulnerabilities. We ran 5 *tlspuffin* fuzzing campaigns on 12 cores that each found 5 of the 7 vulnerabilities within seconds or minutes. The detection of previously known CVEs SDOS1 and SIG takes more effort and is less reliable. We ran 90 fuzzing campaigns for 24h on 1 core each. For each of SIG and SDOS1, 5 of the 90 runs found the vulnerability with a mean time of 564min and 915min, respectively (we explain this higher variance in Section 6.2.2). The full methodology and results can be found in Section C.1.

Result: *tlspuffin found 7 CVEs on OpenSSL and wolfSSL corresponding to DY attacks, including 4 new ones which are reliably found in matters of minutes. Our selection of state-of-the-art bit-level fuzzers and others' fuzzing efforts found none of those 7 vulnerabilities.*

6. One could argue AFLnwe was not designed for stateful PUT. We still consider it for completeness. It turned out it performed similarly to the others.

7. For being able to run those fuzzers and evaluate the achieved coverage (see Section 6.2), we had to resolve several issues and bugs in those projects (added support for wolfSSL, and, more surprisingly, we found and reported 7 bugs in AFLnwe, StateAFL, and TLS-Anvil, including quite critical bugs).

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

TABLE 2: (Re)discovered vulnerabilities with *tlspuffin*. The CVSS scores are the severity scores attributed by NIST. In the Type column, "P" indicates a protocol vulnerability and "M" a memory vulnerability (found by the DY attack). The "New" column indicates whether the vulnerability was first discovered using the *tlspuffin* tool (✓) or rediscovered (✗). **SDOS1** affects *OpenSSL*. The others affect *wolfSSL*.

In Section 6.1.5, we explain why the DY fuzzing approach is key and often necessary to find these vulnerabilities. We first review in Section 6.1.4 some of the (re)discovered bugs[8].

6.1.4. Summary of Vulnerability Descriptions (more in Section B.2). 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 [65] either by introducing a mismatch between the signature algorithm in the Certificate and CertificateVerify messages or by skipping the CertificateVerify message (both are encrypted and require to decrypt the server's response).

CDOS allows servers or MITM to crash *wolfSSL* clients. It is triggered by sending 9 messages, ending with 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 bug in *wolfSSL* servers. Malicious clients can cause a buffer overflow by sending specific Client Hello 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 PSK; and they should omit the supported_groups extension. The trace from Example 5 can be mutated further to produce such an attacking trace. We explain in Appendix C.3 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 Remote Code Execution (RCE).

HEAP is a heap buffer over-read bug on *wolfSSL* servers. It is triggered by sending to a server a malicious

8. We provide the traces found by the tool that trigger those vulnerabilities and comprehensive vulnerability reports (for ours) in [73].

Client Hello message with 25 extensions, notably a dozen key_share extensions.

6.1.5. Advantages of DY Fuzzing.

In Section 2.3, we explained why prior fuzzers are unsuitable for finding DY attacks; we now revisit these reasons in light of our results. *Why were all 7 vulnerabilities missed by the aforementioned campaigns despite intensive fuzzing efforts with state-of-the-art fuzzers?* We also explain why, on the contrary, DY Fuzzing is in a sweet spot for finding them.

(1) Reachability. *(1.a) Harnesses* of most state-of-the-art bit-level fuzzers are often a limitation. We already mentioned the problem of many fuzzers [38, 80, 53] that only send one flight of messages to the PUT, thus excluding all vulnerabilities we found, except **HEAP**. Some other fuzzers, including those we compared with as part of the ProFuzzBench suite, can send multiple flights but only act as clients to fuzz servers and are thus unable to find bugs in clients such as **CDOS**. In contrast, *tlspuffin* can by design act as an attacking client, server, or as a MITM between PUT client and server. *(1.b) Mutations.* Some other vulnerabilities cannot be reached due to the limitations of the used mutations. Standard fuzzers, including those we compared with, rely on bit-level mutations that make it overwhelmingly unlikely that logical transformations of messages will be discovered. 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. The probability for this transformation to happen using random bit-level mutations is upper-bounded by the probability of breaking the encryption scheme. With DY Fuzzing, this adversarial behavior is obtained with a few mutations. All vulnerabilities except **HEAP**, **CDOS**, and **SKIP** require the attacker to apply such cryptographic computations. More generally, all of the 7 vulnerabilities rely on rather complex attacker terms obtained by a simple series of mutations (say t is mutated into t') such that the obtained bitstring $[\![t']\!]_\phi$ is very unlikely to be reached by bit-level mutations from $[\![t]\!]_\phi$. In practice, we empirically established that the fuzzers we compare with did not find **HEAP**, **BUF**, **SDOS1**, or **SDOS2** while *tlspuffin* did, in the same testing environment: same seeds, harness, and detection capabilities (only ASAN).[9]

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

(2) Detection. Classical fuzzers primarily aim at finding memory-related bugs, *e.g.* with ASAN, but their *Objective Oracle* is unable to detect protocol vulnerabilities. In practice, even if a protocol vulnerability, such as the authentication bypass **SIG** and **SKIP**, *was reached*, the aforementioned bit-level fuzzers actually would simply drop this test-case as they would not realize that a policy violation occurred. (The 5 other vulnerabilities can be detected with ASAN.)

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.

9. Doing the same for the others is impossible or challenging due to discussed limitations of the compared fuzzers (harness, detection capabilities).

On State-Machine-Guided Fuzzers. We now compare with related structured fuzzers or testing engines from Section 2.3. **SIG** and **SKIP** were triggered by a state-machine learner [65]; such learners are not fuzzers but a different approach capable of finding *some* DY attacks. They capture adversarial behaviors that drop or repeat whole Handshake messages without the ability to modify their content, except for a limited hard-coded pre-defined messages that can be used. In particular, [65] is unable to reach **CDOS**, **SDOS2**, **BUF**, and **HEAP**. Moreover, such techniques do not feature an *Objective Oracle*; the output is a graph of execution flows that need to be manually interpreted to detect potential attacks. Similar conclusions apply to [14].

6.2. Coverage Comparison

Meaningfully evaluating fuzzers is notoriously difficult and code coverage is an unsuitable metric for this, despite being often used. This is backed up by Klees *et al.* [50]: "*Covering more code intuitively correlates with finding more bugs[...]. But the correlation may be weak, so directly measuring the number of bugs found is preferred.*" (cf. Section 6.1.)

That said, we shall cautiously compute and compare coverage to answer the following **Research Question:** *What specific insights about how tlspuffin relates to state-of-the-art fuzzers can be gained by comparing code coverage?*

6.2.1. Methodology.

Choice of fuzzers and testing engines. We evaluated *tlspuffin* in terms of coverage against our selection of bit-level fuzzers as well as the *combinatorial testing* suite TLS-Anvil [54]. Using ProFuzzBench, we gathered data about 24h fuzzing campaigns against wolfSSL 5.3.0 and OpenSSL 1.1.1j, that are affected by the vulnerabilities from Table 2.

Compute comparable coverage. For the sake of fairness, when comparing different fuzzers, we only chose the seed test cases that can be expressed in all of the compared fuzzers, selected comparable Harness, and ensured line and branch counts are equivalent in all experiments. We also selected a subset of source files to be included in the coverage generation, excluding unit tests, fuzzers, examples and CLIs. We compute coverage over time using the `gcov` tool when executing the test cases of the corpus in the order they were found, which allows us to control the environment and the way the coverage is gathered, which is key to comparability across different fuzzers.

6.2.2. Code Coverage & Feedback Analysis.

We first answer a question from Section 6.1.2 by studying *tlspuffin* coverage: *Why did only 5% of experiments find **SIG** and **SDOS1**?*

To discover **SIG** and **SDOS1**, 3053, respectively 809, mutation tries were required on average (over 100 trials). (Note that the total number of executions in a fuzzing campaign can easily reach 10^8.) Therefore, those vulnerabilities are indeed reachable by *tlspuffin*.

We then found that, once a certain diversity of DY traces is achieved in the *Corpus*, the current code-coverage feedback is "exhausted" and fails to promote DY traces that explore new PUT behaviors but only exercise code already explored (but in different states). To establish this, we analyzed the final corpora of the campaigns that did not discover **SIG** or **SDOS1**. We determined the total code coverage when executing all test cases in the corpus. We tested and observed that applying any subset of the required mutations for finding **SIG** or **SDOS1** on traces from the final corpus did not improve this final code coverage. Therefore, such mutations, when applied one by one, will be discarded (as they are deemed uninteresting): the feedback metric is no longer able to measure progress. This happens when the *Corpus* becomes too large. It is however 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. (Therefore, we recommend running multiple fuzzing campaigns versus only one for a longer period.)

Two key insights can be drawn from this: **Results:** *1. The code coverage is not an ideal feedback metric for DY fuzzings and needs to be improved (left as future work, see Section 7). 2. Code coverage is not a suitable metric to evaluate and compare DY fuzzers since finding new interesting DY traces will not immediately translate to better code coverage.* Another experiment supports this. We compared the achieved coverage when executing the trace triggering SDOS1 versus when executing the seed corpus only. The former covers a marginal amount of new code: no new functions were called (except error functions) and only +120LoC were covered (+0.1% of LoC).

6.2.3. Comparing Coverage of *tlspuffin* vs. State-of-the-art Techniques. We first compared coverage achieved by *tlspuffin* and TLS-Anvil, the best to-date test suite for TLS. A 24h fuzzing campaign on 32 cores with *tlspuffin* achieves coverage on par or a bit less than what TLS-Anvil achieves (-1.6% of LoC were covered for *wolfSSL*). We stress that TLS-Anvil contains hundreds of handwritten tests which are based on TLS-Attacker [70] and were designed to maximize coverage. *E.g.* TLS-Anvil probes targets before testing then tries to do a handshake with every known cipher suite. This dramatically increases the coverage, because tested cipher suites are initialized on the server. Yet, TLS-Anvil found none of the 7 vulnerabilities.

We also compared the coverage of *tlspuffin* against AFLnwe, AFLNet, and StateAFL. As previously mentioned, for the sake of fair comparison, we restricted *tlspuffin* with the limitations of the former fuzzers plugged into ProFuzzBench: exclude clients from the *Harness* and use a smaller *Seed Corpus* (of 2 traces) by removing the 9 *tlspuffin* seeds that were complex to produce for bit-level fuzzers (made easier for *tlspuffin* thanks to our DSL). For all fuzzers and both targets, we executed 10 trials over 24h. For *wolfSSL*, the mean branch coverage achieved by *tlspuffin* across all trials is equal or greater than that of the compared fuzzers, and +33% when *tlspuffin* uses all of its 11 seeds (see Section 4.3). For *OpenSSL*, *tlspuffin* covers 6% less code, and 16% more code with all its seeds.

Manual inspection of the diff between the covered code shows that *bit-level fuzzers and tlspuffin explore different parts of the code*. Bit-level fuzzers are good at exercising server-side code for handling features. A concrete example is cipher suites: since the signature and *Mapper* are currently not exhaustive (but could be made so with more work), *tlspuffin* only implements 3 cipher suites out of 5 for TLS 1.3 and 3 out of 37 for TLS 1.2. Bit-level fuzzers discover them by flipping bits in the Client Hello. However, they are unable to later use such features (*e.g.* a TLS 1.3 extension or a cipher suite) in subsequent messages as it usually requires logical message transformations. The code parts exercised this way can be huge (with a lot of setup and preprocessing work), even though it does not reflect a lot of different behaviors (since the features are actually not used). As a concrete example, consider the file `wolfssl/src/keys.c` which essentially deals with cipher suites. AFLnwe covers 48.6% of the LoC in this file against 28.7% for *tlspuffin*.

Showing the superiority of DY fuzzing at finding DY attacks by comparing coverage. Result: *tlspuffin explores fewer features but is better at actually using those features in subsequent messages.* We obtained empirical evidence supporting this claim by comparing the code exercised by *tlspuffin* when executing detected vulnerabilities versus by long bit-level fuzzing campaigns. We focus on SDOS1 and BUF, as most other vulnerabilities exercise code that is not well harnessed by ProFuzzBench (*e.g.* client-side code). The diff coverage between *tlspuffin* executing those two vulnerabilities versus long bit-level fuzzing campaigns reveals key insights. *(i)* Features like secure renegotiation, required for SDOS1, need modifications under encryption, that were not reached by bit-level fuzzers but explored by *tlspuffin*. *(ii)* BUF requires to produce a Client Hello message with fields computed by applying cryptographic primitives (decrypting application data, hashing, signature and HKDF). The code parsing this field is not exercised by bit-level fuzzers, which are unable to perform such logical transformations. (Our full coverage evaluation can be found in Section B.3.)

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 6.2.2 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 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 $[\![\cdot]\!]$ when possible, see Remark 3) 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 to the underlying protocol DY model. Finally, recent work allows verifying privacy properties [24], *e.g.* anonymity, that provides a challenging opportunity for extending this work.

Another direction is to augment the attacker capabilities by the addition of more mutations (*e.g.* modifying the channels and their configurations) and operators in Σ (*e.g.* systematic dummy values for each type).

Finally, the use of *differential fuzzing*, *i.e.* executing a test-case on multiple PUTs and labeling it as an objective (attack candidate) if the outputs differ, could simplify our *Objective Oracle*.

Apply to more targets. We also want to apply DY fuzzing to more targets. Candidates for TLS PUT could be Google's BoringSSL, Microsoft's closed-source Schannel, or the open-source Mbed TLS. We also plan to improve *sshpuffin* for SSH. DY fuzzers should be applied to 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. A related long-term future work is to partially automate the construction of the *Mapper*, *e.g.* by static analysis of crypto libraries.

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 were later added during a collaboration with Trail of Bits from New York, USA. This work has also been partly supported by the ANR JCJC project ProtoFuzz (ANR-22-CE48-0017), ANR Research and teaching chair in AI ASAP (ANR-20-CHIA-0024) with support from the region Grand Est.

We also thank Alexander Knapp and our anonymous reviewers for their comments on early drafts 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] *aflnwe*. https://github.com/thuanpv/aflnwe.

[4] 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.

[5] L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. Basin, and P. Müller. “Sound verification of security protocols: From design to interoperable implementations”. In: *Symposium on Security and Privacy (SP).* IEEE. 2023.

[6] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert. “NAUTILUS: Fishing for Deep Bugs with Grammars”. In: *Network and Distributed System Security Symposium (NDSS).* The Internet Society, 2019.

[7] 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.

[8] 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”. In: *Symposium on the Foundations of Software Engineering (ESE/FSE).* ACM, 2019.

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

[10] 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.

[11] 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: *USENIX Security Symposium.* 2021.

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

[13] 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 Soft-

ware Analysis Platform". In: *Communications of the ACM* 64.8 (2021).

[14] 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.

[15] 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 (EuroSP)*. IEEE. 2021.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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).

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

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

[23] S. Y. Chau, O. Chowdhury, M. E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li. "SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations". In: *Symposium on Security and Privacy (SP)*. IEEE, 2017.

[24] 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.

[25] 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).

[26] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[27] 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.

[28] *CVE - CVE-2009-3555*. https://www.cve.org/CVERecord?id=CVE-2009-3555.

[29] *CVE - CVE-2014-0160*. https://www.cve.org/CVERecord?id=CVE-2014-0160.

[30] *CVE - CVE-2014-1266*. https://www.cve.org/CVERecord?id=CVE-2014-1266.

[31] *CVE - CVE-2016-2108*. https://www.cve.org/CVERecord?id=CVE-2016-2108.

[32] L. Daniel, E. Poll, and J. de Ruiter. "Inferring OpenVPN State Machines Using Protocol State Fuzzing". In: *European Symposium on Security and Privacy Workshops*. IEEE, 2018.

[33] J. De Ruiter and E. Poll. "Protocol State Fuzzing of TLS Implementations". In: *USENIX Security Symposium*. 2015.

[34] 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.

[35] 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.

[36] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on information theory* 29.2 (1983).

[37] N. Drucker and S. Gueron. *Selfie: Reflections on TLS 1.3 with PSK*. ePrint IACR 347. 2019.

[38] 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)*. 2020.

[39] 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.

[40] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky. "Analysis of DTLS Implementations Using Protocol State Fuzzing". In: *USENIX Security Symposium*. 2020.

[41] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky. "Analysis of DTLS Implementations Using Protocol State Fuzzing". In: *USENIX Security Symposium*. 2020.

[42] *Fuzzowski Network Fuzzer*. https://github.com/nccgroup/fuzzowski.

[43] 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". In: *USENIX Security Symposium*. 2020.

[44] Google. *honggfuzz*. https://honggfuzz.dev/.

[45] A. Hazimeh, A. Herrera, and M. Payer. "Magma: A Ground-Truth Fuzzing Benchmark". In: *Proc. ACM Meas. Anal. Comput. Syst.* 4.3 (2020).

[46] E. Hoque, H. Lee, R. Potharaju, C. Killian, and C. Nita-Rotaru. "Automated Adversarial Testing of Unmodified Wireless Routing Implementations". In: *IEEE/ACM Trans. Netw.* 24.6 (2016).

[47] M. E. Hoque, O. Chowdhury, S. Y. Chau, C. Nita-Rotaru, and N. Li. "Analyzing Operational Behavior of Stateful Protocol Implementations for Detecting Semantic Bugs". In: *International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017.

[48] S. R. Hussain, I. Karim, A. A. Ishtiaq, O. Chowdhury, and E. Bertino. "Noncompliance as Deviant Behavior: An Automated Black-Box Noncompliance Checker for 4G LTE Cellular Devices". In: *Conference on Computer and Communications Security (CCS)*. ACM, 2021.

[49] I. Karim, A. Ishtiaq, S. Hussain, and E. Bertino. "BLEDiff : Scalable and Property-Agnostic Noncompliance Checking for BLE Implementations". In: *Symposium on Security and Privacy (SP)*. IEEE, 2023.

[50] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. "Evaluating Fuzz Testing". In: *Conference on Computer and Communications Security (CCS)*. ACM, 2018.

[51] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru. "Turret: A Platform for Automated Attack Finding in Unmodified Distributed System Implementations (ICDCS)". In: *International Conference on Distributed Computing Systems*. IEEE, 2014.

[52] X. Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009).

[53] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html.

[54] M. Maehren, P. Nieting, S. Hebrok, R. Merget, J. Somorovsky, and J. Schwenk. "TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries". In: *USENIX Security Symposium*. 2022.

[55] 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).

[56] 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.

[57] 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).

[58] B. Möller, T. Duong, and K. Kotowicz. *This POODLE Bites: Exploiting The SSL 3.0 Fallback*. Tech. rep.

[59] M. Musuvathi and D. R. Engler. "Model Checking Large Network Protocol Implementations". In: *Conference on Symposium on Networked Systems Design and Implementation (NSDI)*. 2004.

[60] R. Natella. "StateAFL: Greybox Fuzzing for Stateful Network Servers". In: *Empirical Softw. Engg.* 27.7 (2022).

[61] R. Natella and V.-T. Pham. "ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing". In: *International Symposium on Software Testing and Analysis*. ACM, 2021.

[62] K. G. Paterson and T. van der Merwe. "Reactive and Proactive Standardisation of TLS". In: *Security Standardisation Research (SSR)*. 2016.

[63] 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.

[64] 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.

[65] 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.

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

[67] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz. "Nyx-Net: Network Fuzzing with Incremental Snapshots". In: *European Conference on Computer Systems (EuroSys)*. ACM, 2022.

[68] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *USENIX Technical Conference*. 2012.

[69] K. Serebryany. "OSS-Fuzz - Google's continuous fuzzing service for open source software". In: *USENIX Security Symposium*. 2017.

[70] J. Somorovsky. "Systematic Fuzzing and Testing of TLS Libraries". In: *Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[71] J. Song, C. Cadar, and P. Pietzuch. "SymbexNet: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications". In: *IEEE Trans. Softw. Eng.* 40.7 (2014).

[72] 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*". In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 2016.

[73] *tlspuffin*. https://github.com/tlspuffin. 2023.

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

[75] 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.

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

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

[78] WolfSSL Project. *Fuzz Testing*. https://www.wolfssl.com/fuzz-testing/.

[79] M. Yahyazadeh, S. Y. Chau, L. Li, M. H. Hue, J. Debnath, S. C. Ip, C. N. Li, E. Hoque, and O. Chowdhury. "Morpheus: Bringing The (PKCS) One To Meet the Oracle". In: *Conference on Computer and Communications Security (CCS)*. ACM, 2021.

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

[81] 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.
- v1.1, August 29, 2023: Main modifications:
  - Explicit distinction between *DY attacks triggering protocol vulnerabilities* and *DY attacks triggering memory vulnerabilities*;
  - Additional related work in Section 2.3;
  - New evaluation of *tlspuffin* (Section 6) to include a quantitative (coverage) and qualitative comparison with AFLnwe, AFLNet, StateAFL, and TLS-Anvil.
- v1.2, December 1st, 2023: Modifications that we made for the camera-ready version of this report, which will appear at IEEE Security and Privacy 2024. Main modifications:
  - logical attacks are now called DY attacks to disambiguate (see Introduction);
  - various clarifications and improvements.

Appendix B. Additional Content

B.1. 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** [72] requires a dedicated protocol implementations and significant effort. As an illustration, proofs for the record layer of QUIC [35] 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 [34] and for TLS 1.2 the proof (using *F7*) does not cover the full handshake [18]. 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 [4]. No cryptographic 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** [81], written in *F**, start being deployed in production systems, such examples are still the exception, and cover the cryptographic library, not the protocol.

The *DY** [15] verification framework 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. Another recent technique combines DY model verification and code verification using a dialect of separation logic [5] to provide security guarantees to protocol implementations. However, proving an existing implementation (currently in Go or Java) requires a large amount of work; *e.g.* the portion of 608 verified LoC in WireGuard require 3936 LoC for the specifications and proof annotations.

Finally, program verification of existing code, for instance in C, has been made possible with tools like *Frama-C* [13] combined with security proofs in tools like *Easycrypt* [10], and secure compilers like *CompCert* [52]. However, this approach does not currently scale beyond simple cryptographic primitives, excluding cryptographic protocol implementations.

B.2. Vulnerability descriptions

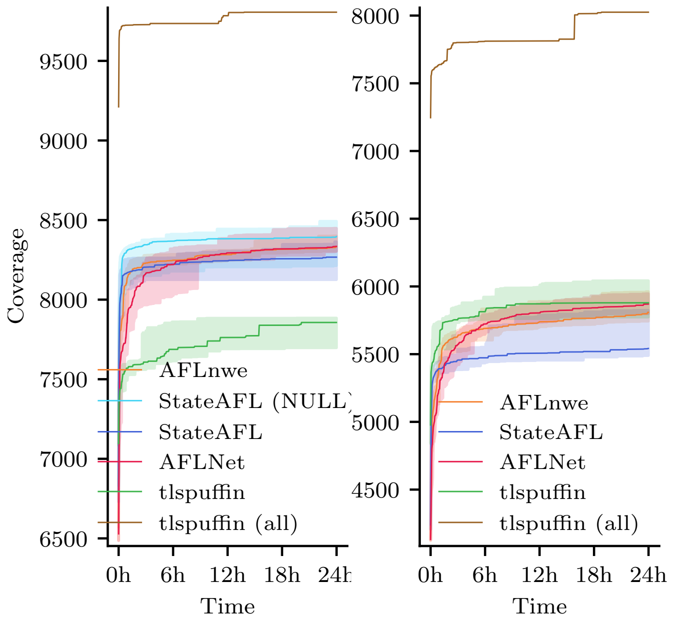
The **SDOS1** vulnerability allows malicious clients to crash *OpenSSL* servers during TLS 1.2 renegotiation by omitting the signature_algorithms extension but including a signature_algorithms_cert extension.

SIG and **SKIP** are two bugs allowing client authentication bypass in *wolfSSL* servers [65]. 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. *tlspuffin* is able to produce such logical message modifications and to automatically detect an authentication bypass through its *Objective Oracle*.

SDOS2 allows clients or MITM to crash *wolfSSL* servers. A client can resume 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.

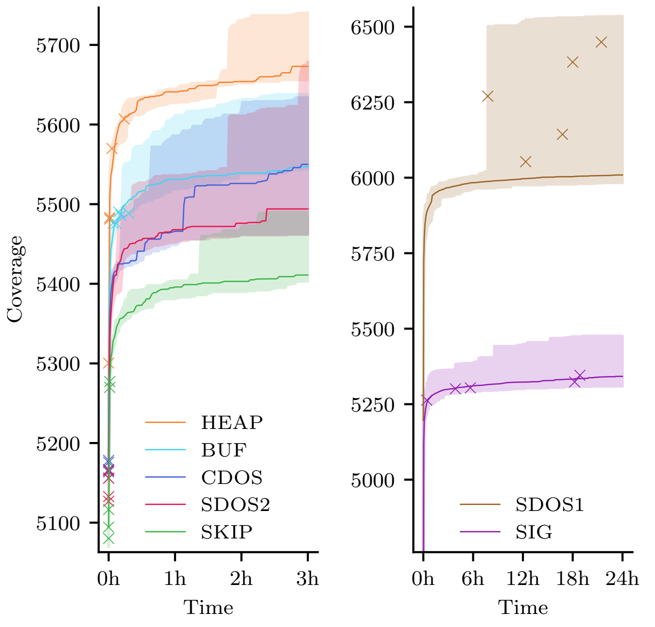
CDOS is a bug allowing servers or MITM to crash *wolfSSL* clients. An attacker triggers this bug by sending a large NewSessionTicket message (> 256 bytes) to a client with a non-empty session cache, who will then free a pointer to non-allocated memory and crash.

BUF is a stack buffer overflow bug in *wolfSSL* servers. Malicious clients can cause a buffer overflow by sending



(a) Part II - OpenSSL 1.1.1j (b) Part II - wolfSSL 5.3.0

Figure 2: Branch coverage comparison between tlspuffin, AFLnwe, AFlNet and StateAFL. For each fuzzer and target 10 trials have been executed, except for tlspuffin (all) which serves solely for the comparison with TLS-Anvil from Part I. The error bar depicts the area limited by the worst and best performing trial. The opaque line shows the median of all trials. tlspuffin (all) shows a fuzzing campaign which contains all of the 11 seeds developed for *tlspuffin* (see Section 6.2.3).



(a) Group A: 5 experiments per vuln., each on 12 cores (b) Group B: 90 experiments per vuln., each on 1 core

Figure 3: Code edge coverage of *tlspuffin* 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.

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 C.3 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.

B.3. Coverage Evaluation (continuing Section 6.2)

B.3.1. Part I: Comparing coverage of *tlspuffin* vs. bit-level fuzzers. We compared the coverage of *tlspuffin* against AFLnwe [3], AFLNet [63], and StateAFL [60], which are all based on the bit-level AFL fuzzer [38]. All three fuzzers send messages over TCP to the target. AFLNet and StateAFL are stateful, which means that they send multiple flights of messages. The output of the target is analyzed between each flight and influences future flights. For comparability we restricted *tlspuffin* to fuzz only TLS 1.2 and 1.3 servers. We recorded TLS 1.2 and 1.3 input seeds for AFLnwe, AFLNet and StateAFL using WireShark. Additionally we recorded a seed with a NULL cipher suite for OpenSSL for evaluation with StateAFL. For all fuzzers and both targets, we executed 10 trials over 24h. In total this yields 70 fuzzing campaigns which are shown in Figure 2.

The branch coverage achieved by tlspuffin across all trials is on par with the compared fuzzers, and greater when tlspuffin uses all of its 11 seeds, which was to be expected.

It is noteworthy, that in our evaluation AFLnwe and AFLNet outperformed StateAFL. This is surprising given the fact that AFLnwe is not stateful as it sends only a single flight of messages, whereas StateAFL is stateful. We suspect that this is because we included a TLS 1.3 seed, which is encrypted right after the first Client Hello.

It is unlikely that any of the AFL based fuzzers fuzz beyond the Client Hello message. The reason for the increase in coverage over time can be attributed to features which are discovered by flipping bits in the Client Hello. For instance, flipping a bit in the proposed cipher suites will increase coverage substantially (see Section 6.2.3). Therefore, even though coverage is increasing over time the AFL based fuzzers likely do not explore state-space which is relevant for DY bugs. We provide more insights in Appendix C.

B.3.2. Part II: Comparing coverage of *tlspuffin* vs. comprehensive test-suite (TLS-Anvil). We compared *tlspuffin* fuzzing *OpenSSL* and *wolfSSL* on 32 cores over 24h with TLS-Anvil. We made sure that *OpenSSL* and *wolfSSL* TLS 1.2 and 1.3 servers were tested.

TLS-Anvil, respectively *tlspuffin*, reached 25.8%, respectively 19.9% line coverage on *OpenSSL* and 29.0%, respectively 24.2% on *wolfSSL*. More detailed figures about line, branch and function coverage are given in Table 3.

	OpenSSL		*wolfSSL*	
	TLS-Anvil	*tlspuffin*	TLS-Anvil	*tlspuffin*
l_p	25.8%	19.9%	29.0%	24.2%
l_a	28074	21664	23421	19581
b_p	20.2%	16.0%	19.9%	16.9%
b_a	12363	9806	9434	8025
f_p	30.1%	23.9%	31.9%	27.0%
f_a	2370	1885	1225	1037

TABLE 3: Code coverage TLS-Anvil vs *tlspuffin*. l_x: lines, b_x: branches, f_x: functions, p: percentage, a: absolute

tlspuffin currently uncovers less code than TLS-Anvil. The reason for this is that TLS-Anvil contains many handwritten tests which are based on TLS-Attacker. The years of development which went into TLS-Anvil it is likely that more features of TLS are covered. TLS-Anvil also probes targets before testing then, which tries to do a handshake with every known cipher suite. Such probing will increase the coverage dramatically, because tested cipher suites are initialized on the server. *tlspuffin* currently does not do this so it is expected that the coverage is less than TLS-Anvil. Nonetheless, the current coverage of *tlspuffin* is promising given its just recent development.

B.4. Reliability, Feedback, and Performance Evaluation

Finally, to complete the evaluation of our first *tlspuffin* implementation, we answer the following **Research Questions:** *How reliable is our tool at finding bugs? When our tool does not reliably find a bug, what are the reasons for this? How efficient is tlspuffin?*

To answer these questions, we conducted a quantitative and qualitative evaluation of our benchmarks, *i.e.* findings bugs from Table 2. The full methodology and results are presented in Section C.1.

We summarize in Figure 3 the results showing the time-to-find the different vulnerabilities.

Appendix C. Results and Comparison

C.1. Reliability, Feedback, and Performance Evaluation

Finally, to complete the evaluation of our first *tlspuffin* implementation, we answer the following **Research Questions:** *How reliable is our tool at finding bugs? When our tool does not reliably find a bug, what are the reasons for this? How efficient is tlspuffin?*

To answer these questions, we conducted a quantitative and qualitative evaluation of our benchmarks, *i.e.* finding bugs from Table 2.

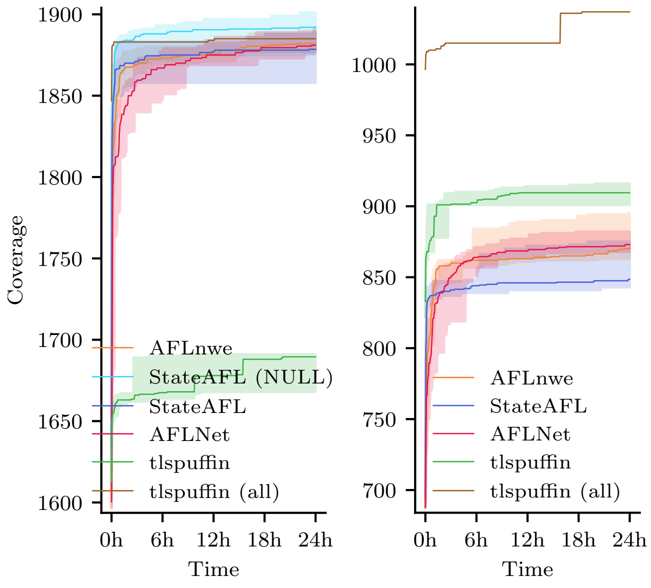
C.1.1. Methodology. The evaluation was conducted on a server with 500 GB of RAM and 2 `AMD EPYC 7F521` processors with 16 physical cores each (for a total of 64 logical cores). (Note that such a large computation power and amount of RAM are not mandatory to use *tlspuffin*.) The experiments conducted in this evaluation are completely reproducible. We created bash scripts [73], which download statically linked binaries from our CI and execute experiments.

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.[10] We created two groups of vulnerabilities: Group A (listed in Figure 3a) are those found in a matter of minutes, and group B (listed in Figure 3b) 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.

C.1.2. Experiment Results. In Figure 3a (Group A) and 3b (Group B), we present the code edge coverage and vulnerability discovery times.

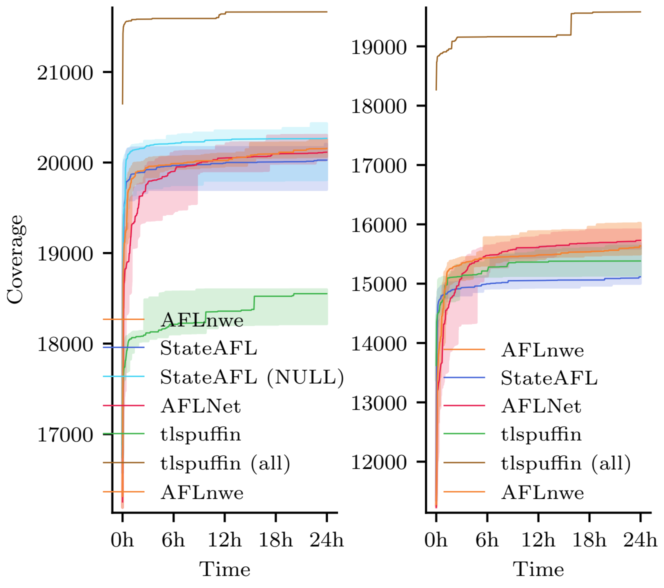
Out of a total of 180 experiments in group B, we discovered SIG and SDOS1 only five times each. In group A, all bugs were *always* found within minutes after launching the campaign, whereas in group B discovering the bugs took hours. The mean time until SIG was discovered was 564min ($\pm$ 510min). For SDOS1, the mean time was 915min ($\pm$ 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.

10. For the sake of evaluation reliability, we deliberately patched other vulnerabilities without impacting the unique vulnerability we expected to find. The advantages of this are twofold. (i) Other known and existing bugs do not impact the performance of the fuzzer. This is otherwise a major factor of unpredictability and variability. (ii) Analyzing and triaging the found bugs is simple, because there is less variety in the bugs found.



(a) Part II - OpenSSL 1.1.1j (b) Part II - wolfSSL 5.3.0

Figure 4: Function coverage



(a) Part II - OpenSSL 1.1.1j (b) Part II - wolfSSL 5.3.0

Figure 5: Line coverage

C.1.3. 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.)

We provide additional results in a dedicated section Section C.2.

C.1.4. 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) [68]. 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 [68]. 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* allows parallel processing, 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 C.4.

C.1.5. Mutations Ablation Study. We conducted an ablation study for the mutations. Namely, we re-ran all of the fuzzing campaigns of Task A and Task B (from Section C.1.2) with *tlspuffin* with one less mutation, restricted to the vulnerabilities **SDOS1**, **SDOS2**, **BUF**, and **HEAP**. We measured whether *tlspuffin* with one mutation disabled was still able to find the target vulnerability, and if it is the case, how much PUT executions it needed before finding it. The results of this experiment are summarized in Table 4.

As shown in the Table, each of the mutation, except REMOVE-AND-LIFT, provides a measurable advantage to *tlspuffin* highlighted in bold and red font: either by allowing it to find a vulnerability not found otherwise, or to reduce the effort to find it (by at least a significant factor). We decided to keep REMOVE-AND-LIFT as it could still be useful if reducing the size of a list, *e.g.* a list of TLS extensions, was necessary to find a bug, even if none of the 7 we found required this. Moreover, it can be applied a bounded number of times only on a trace.

C.2. Mutation Benchmark

We provide additional data about the mutation evaluation from Section C.1.3. 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

	all	*Skip*	*Repeat*	*Swap*	*Generate*	*Replace-Match*	*Replace-Reuse*	*Remove-and-Lift*
SDOS1	2,3E+07	✗	✗	2,4E+07	3,6E+06	1,6E+07	1,7E+07	2,1E+07
SDOS2	6,3E+03	8,7E+03	4,0E+06	2,7E+04	2,4E+03	**1,2E+04**	2,0E+03	4,4E+03
BUF	9,8E+04	6,1E+04	3,4E+04	9,1E+04	**2,8E+05**	1,1E+05	5,1E+04	3,5E+04
HEAP	1,3E+05	1,9E+05	4,5E+04	**2,0E+05**	1,1E+05	**1,9E+05**	**3,0E+05**	1,8E+04

TABLE 4: For each column, we disable the corresponding mutation and compute the number of PUT executions before *tlspuffin* finds each of the vulnerability (in each row). For the first column (all), all mutations are enabled (baseline). We compute the average for at least three trials. ✗ denotes that the vulnerability was not found. Notes that the number of executions before finding a bug is subject to variance, but, is at least independent of the stress on the machine running the benchmark (which is not the case of the time to find the vulnerability). Moreover, the numbers can be lower than the one in the column ”all” due to the expected variance but also due to the fact that removing mutations might speed up the fuzzer when the mutation is not needed for finding the vulnerability. We only tested vulnerabilities that are reliably found in that setting.

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.

C.3. Notes on Triaging BUF

As explained in Section E.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 [73]).

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 [73]) indicates that two maliciously formed ClientHello messages, which contain multiple duplicate cipher suites, are responsible for the bug (mentioned Section 6.1.3 and detailed in Remark 4 below).
3) Dynamically analyze the bug: Using the `gdb`/`lldb` 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 4 (**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*

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 5: 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.

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 Proof-of-Concept (PoC). Finally, we could use the bitstring extractions to produce a `netcat` command as PoC.

C.4. Detailed Performance Results and Plots

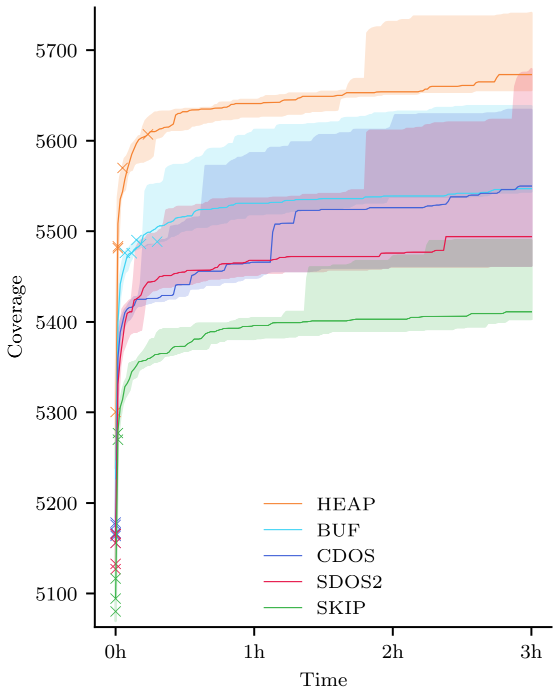
See Table 5 and enlarged version of Figure 3 in Figure 6.

C.5. 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[11] 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.

11. https://openwrt.org/advisory/2022-10-04-1 (warning to reviewers: the page content at this link acknowledges the authors)



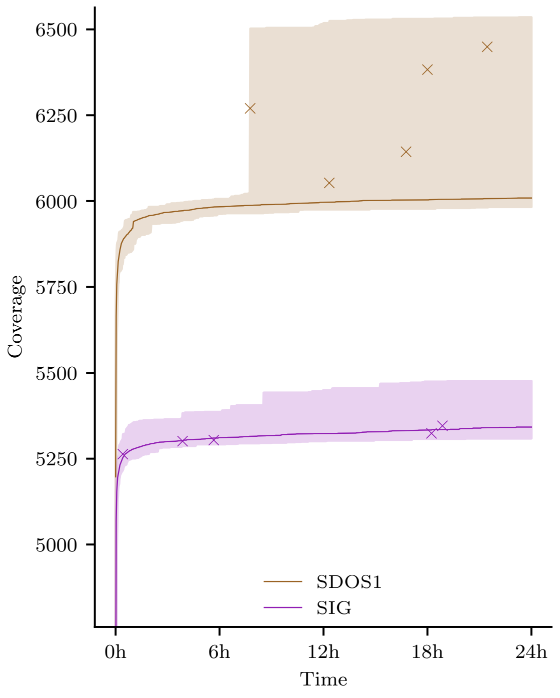


Figure 6: Larger version of Figure 3.

```
let rsa_certificate = term! {                      1
  fn_certificate13(...)                            2
};                                                 3
                                                   4
let certificate = term! {                          5
  fn_encrypt_handshake(                            6
    (@certificate_rsa),                            7
    (fn_sh_transcript(((server, 0)))),             8
    (fn_server_share(((server, 0)))),              9
    fn_seq_0,                                      10
    ...                                            11
  )                                                12
};                                                 13
```

Figure 7: Example usage of the term DSL

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.

Appendix D. DY Fuzzing

D.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 7, 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 an atom 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 [73].

D.2. Notes on Mutations

D.2.1. 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 untyped (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.

D.2.2. Mutation Redundancy.

Note that the result of the *Swap* mutation between two terms whose one is a sub-term 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++ [38].

D.2.3. 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 $f(t_1, \dots, t_k)$ with $f'(t_1, \dots, t_k)$ such that $f(t_1, \dots, t_k) \in \mathsf{dom}([\![\cdot]\!]_\phi)$ (the computation succeeds) but $f'(t_1, \dots, t_k) \notin \mathsf{dom}([\![\cdot]\!]_\phi)$ (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 E. Implementation: The *tlspuffin* Fuzzer

E.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, which is at the logical level.

E.2. Implementations Challenges

Queries. We had to tackle a 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. For *wolfSSL*, we 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. As detailed in Section 4.2, some claims could also be extracted without any modification or even access to the PUT.

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.

Transcript Extraction. When running earlier versions of *tlspuffin*, we noticed that some attacker terms could get huge for the trace to be executed gracefully (sometimes with >10k operators per term). 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 reduces the likelihood of finding mutations that also mutate the transcript hash accordingly and dramatically increases the size of terms in traces, since to each transcript hash corresponds a (large) term leading to many duplicated terms.

To address those problems, we decided to give the attacker the possibility to use in attacker terms a shortcut $\texttt{hashTr} @ c$ that refers to the hash transcript as c would compute it at this time of the execution. We then let *tlspuffin* replaces those with the dynamically computed transcript hashes. This dramatically reduces duplicate transcript terms and term sizes. This is without loss of generality since the transcript being hashed is made of messages sent in cleartext, so the attacker knows it already. We are using the same implementation methods that we use for claim extraction for transcript extraction.

E.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 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 C.3 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.

***tlspuffin* as an Analysis Framework.** As for *TLS-Attacker* [70], *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.