Max Ammann, Lucca Hirschi, Steve Kremer

To cite this version:
Max Ammann, Lucca Hirschi, Steve Kremer. DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing. 45th IEEE Symposium on Security and Privacy, 2024, San Francisco (CA, USA), United States. hal-04318710v2

HAL Id: hal-04318710
https://inria.hal.science/hal-04318710v2
Submitted on 5 Dec 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

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

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.

Logical attacks and formal verification. Since the 1980s [37], the formal methods community has identified and mathematically defined the Dolev-Yao (DY) attacker and the corresponding class of logical attacks [11]. 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 $\text{enc}(m, k)$ with the right key yields the plaintext: $\text{dec} (\text{enc}(m, k), k) = m$). 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 [29][17][27][28][19][59][20][8][58]. Similar attacks impacted many other protocols such as WPA2 [75][76], credit card payment systems [13][12], 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 [63][78]).

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: [31][15][34][41][29][59][22][66]. 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, preclud-
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 [58] and is now the gold standard for testing security software. It has become a key part of software development practices; e.g., Google [9], 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 [11]. 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 [11].

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) [69]) 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.

tlsuffin: 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 fuzzing module puffin is built on top of the fuzzing library LibAFL [40], where we re-implemented most of the fuzzing components. On top of puffin, we build protocol-dependent fuzzers: tspluffin for TLS and the preliminary sshpuffin for SSH. Third, we connected PUTs to the fuzzers: OpenSSL, LibreSSL, and wolfSSL to tspluffin and libssh to ssphpuffin. 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 tspluffin 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. \textit{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 [51]).

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.
2) We propose \textit{tlspuffin}: a full-fledged, modular, and efficient implementation in Rust of this fuzzer design.
3) We evaluate and compare \textit{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 \textit{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 [67] 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) 

\textbf{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) \textbf{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 \textit{handshake} protocol negotiates cipher suites, authenticates the endpoints, and establishes a shared, session key; then, the \textit{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 [67], Appendix A]. We give a bit more details about the handshake protocol as we will use it for illustration throughout the paper.

\textbf{Key Exchange and Authentication}. The handshake protocol starts with two \textit{key exchange} messages. The client sends a \textit{ClientHello} message containing proposed cipher suites, and either an ephemeral (EC)DH key share or a PSK.

The server replies with a \textit{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 \textit{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 \textit{Certificate}, e.g., a X.509 certificate, and a \textit{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 \textit{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.

\textbf{Session resumption}. At the end of the handshake the server transmits a \textit{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.

\textbf{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 [39][54][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 \textit{Corpus}, i.e. set of ”interesting” test cases, whose execution by the PUT will provide \textit{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: \textit{generation-based fuzzers} utilize a specification of the input space (e.g. a grammar) while \textit{mutation-based fuzzers} leverage (often random) mutations to generate new test cases from the current \textit{Corpus} and add those to this corpus if deemed interesting according to the obtained feedbacks. Finally, an \textit{Objective Oracle} checks \textit{Security Policies} when executing the test cases to detect security violations in the PUT, such as a buffer overflow.

We are mostly using the terminology of \textit{LibAFL} [40], which is a state-of-the-art, modular, efficient fuzzing framework we build on. In its default configuration, \textit{LibAFL} implements \textit{evolutionary fuzzing}, which is mutation-based
and grey-box, i.e., 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 [34, 77, 23, 43]. Such testing methodologies are adequate to find safety vulnerabilities with potential security implications (e.g., HeartBleed [31] and CloudBleed [10]), 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 [39, 81, 54, 24, 80, 7] 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, 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 [39, 81, 54, 64, 77, 23, 45, 43, 52, 47, 55, 71, 50, 61, 72] or/and based on mutations that can only modify a hard-coded selection of message fields [52, 55, 71, 50, 72]. 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 \textit{stateful fuzzing approach} [10] and use input-output protocol Finite State Machines (FSM) as a behavioral abstraction of the PUT and use \textit{differential fuzzing} to detect potential bugs [50, 49] or manually inspect the inferred FSM [66, 34, 15, 41, 65]. 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 [72, 50]) and thus suffer from the limitation (1b); the same applies to many other testing methodologies and fuzzers [48, 68, 33, 49, 60].

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 A.1 that verification methods targeting implementations of cryptographic protocols (e.g. [73, 56, 35, 19, 4, 82, 16, 14, 11, 53, 9]), 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: \textit{scalability} to whole large protocols and, more importantly, ability to \textit{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

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.
and Yao [37] that is today the basis for numerous verification techniques [11] and real-world security analysis [17] [27] [28] [13] [44] [18] of protocol designs (but not of implementations). We now recall some preliminary basic definitions in this model and refer the curious reader to [26] [21].

3.1. Term Algebra

In DY models, messages are described using a term algebra. For example, the term 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, sdec(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 $\Sigma$ 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: $\text{senc} \in \Sigma$ 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 $\Sigma$ that contains operators $\text{sign}(\cdot, \cdot)$, $\text{checksign}(\cdot, \cdot)$, $\text{pk}(\cdot)$, and an atom 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 $\text{CHello}(\text{sid}, \text{cs}, \text{ext})$ represents a ClientHello message where $\text{sid}$ is a session identifier, $\text{cs}$ and $\text{ext}$ are terms that encode the list of proposed cipher suites, respectively. Extensions. Note that $\text{CHello}()$ models the formatting but does not provide any cryptographic protections. We therefore suppose that we also have three operators $\pi_1, \pi_2, \pi_3$ 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 $\text{checksign}(\text{sign}(x, y), \text{pk}(y)) = \text{true}$ with $x, y \in \mathcal{V}$. This does indeed model signature verification as the public key used for verification $\text{pk}(y)$ must match the signature key $y$, which may be instantiated with any term. To illustrate the black-box cryptography assumption, we emphasize that this would be the only algebraic property satisfied by checksign, and hence the only way to correctly verify a signature in this model is to use a matching key. This models a strong form of the unforgeability cryptographic assumption.

In DY protocol verification, an equational theory is required to reason about protocols [26] [27]. 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 4.7).

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

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

Intuitively, the output action $\text{out}(c, x)$ 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 $\text{in}(c, t)$ 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.

Example 3. We assume two channels $c$ and $s$ of respectively a TLS 1.3 client and server. Consider the trace $A := \text{out}(c, x).\text{in}(s, t) \in A$ 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. 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 $t = \text{someError}$, 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 $t = ch(0, \pi_2(x), \pi_3(x))$ would correspond to the same message but replacing the session identifier sid by the atom 0.

3. Attacker terms are often called “recipes” in the literature.
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 [26] [21]) or into concrete semantics, as done in our DY fuzzing approach and detailed in Section 4.

Recall that each channel \( c \in C \) 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 \( S \) the set of all local states. The global state is defined by a partial function:
\[
\phi : V \rightarrow M.
\]
The domain of \( \phi \) contains variables referring to previous outputs from this attacker state. The set of such partial functions \( \phi \) is denoted by \( \Phi \).

We suppose a generic specification of honest agents by the means of two abstract partial functions:
\[
\begin{align*}
\text{output} : & \quad S \rightarrow S \times M \\
\text{input} : & \quad S \times M \rightarrow S.
\end{align*}
\]
Intuitively, when an agent \( c \) is in state \( s_c \), \( \text{output}(s_c) \) 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 \( s'_c := \text{input}(s_c, m) \). Note that those functions are partial as honest agents might block.

In order to transform terms in \( T \) (and notably attacker terms) into messages in \( M \) we associate to each operator \( f \in \Sigma \) an interpretation \( [f] : M^i \rightarrow M \) when \( f \) has arity \( i \). Given \( \phi \in \Phi \), we inductively lift \([\cdot] \) to terms as follows:
\[
\begin{align*}
[f(t_1, \ldots, t_i)]_{\phi} & \equiv [f]([t_1]_{\phi}, \ldots, [t_i]_{\phi}) \\
[x]_{\phi} & \equiv \phi(x) & & \text{if } x \in V \cap \text{dom}(\phi)
\end{align*}
\]
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 \( \phi \):
\[
\begin{align*}
\text{output}(s(c)) = (s', m) & \quad \text{and} \\
\text{input}(s(c)) = (s'_c, \phi) & \quad \text{when output}(s(c)) = (s', m)
\end{align*}
\]
when \( [t]_{\phi} = s' \). Intuitively, an action \( \text{output}(c, x) \) updates the state of the agent \( c \) and records the message \( m \) output by \( c \) in the attacker’s state \( \phi \). 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 \( \phi \).

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):
\[
(s_0, \emptyset) \xrightarrow{a_1} (s_1, \phi_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (s_n, \phi_n).
\]

**Remark 2.** As is standard in DY protocol verification, honest agents are usually specified in a formal language such as the applied \( \pi \)-calculus [27] (or multiset rewriting rules [22]). In that case, states in \( S \) correspond to \( \pi \)-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 \( [\_] \) is simply the identity function, i.e., the operators are uninterpreted. The formal model yields output 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 \( [\_] \) 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 \([\_] \) transforms terms (\( T \)) 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. 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 \( c(m_1, \ldots, m_i) \) where \( c \) is a symbol with an arity \( i \) and \( m_1, \ldots, m_i \in M \) are messages. We illustrate this notion on two particular kinds of claims that we will use throughout the paper. Let \( pk, pk_{peer}, m \in M \).

- **Agreement claims** \( \text{Agr}(pk, pk_{peer}, m) \) 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** \( \text{Run}(pk, pk_{peer}, m) \) 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 claims : \( S \rightarrow P(C) \) 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
\[
(s_0, \emptyset) \xrightarrow{a_1} \cdots \xrightarrow{a_n} (s_n, \phi_n)
\]
we define the corresponding trace of claims \( C \) as
\[
C := \text{claims}(s_0) \cup \text{claims}(s_1) \cup \cdots \cup \text{claims}(s_n).
\]
\(C_0, \ldots, C_n\) where \(C_i = \bigcup_{c \in \text{dom}(s_i)} \text{claims}(s_i(c))\) corresponds to all claims extracted from the \(i\)th state. Claims are therefore positioned, and we say Agr(pk, pk', m) @ i is true in \(C\) if Agr(pk, pk', m) \(\in C_j\). Our logic for expressing properties is reminiscent to the one used in the Tamarin prover [57].

**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:
\[
\forall pk, pk', m, i. \text{Agr}(pk, pk', m) @ i \Rightarrow \exists j. \text{Run}(pk', pk, m) @ j \wedge j < i
\]
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 \(A\). 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 \(T\), 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 \(C\). The PUT sends back bitstrings that get added to the attacker’s state \(\phi\). 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.

Benefitting 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 9).

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 \([f]\). 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 \(f \in \Sigma\), the Mapper uses the corresponding function in SignatureLib implementing this function for computing \([f]\). \([f]\) is a partial function since its computation might fail or return an error. For an atom \(f_0\), we define \([f_0]\) 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 \(A = a_1 \cdots a_n \in A\), it will do the following.

**Agents creation.** The Harness creates a new PUT session \(s_c\) for each channel \(c \in C\) 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 \(s_c \in S_0\), 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, $\mathcal{I}$ 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 $s'_c := \text{input}(s_c, m)$ 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, $(s'_c, m) := \text{output}(s_c)$ 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 claims : $\mathcal{S} \rightarrow \mathcal{P}(\mathcal{C})$ (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 [5, F.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 for all possible expected protocol configurations. When evaluating our fuzzer we solely use happy flows and no attack traces as seeds.

For instance for TLS 1.3, we consider two different honest agents, Alice and Bob. We consider seeds for each of the following happy flows: (i) full handshake between Alice and Bob without decrypting messages (DY attacker acting as a passive MITM), (ii) full handshake between the DY attacker, acting as an honest server, and Alice acting as client, (iii) full handshake between the DY attacker, acting as an honest client, and Bob acting as server, and (iv) the happy flow 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).

4.4. 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 [5, C.1.5].

<table>
<thead>
<tr>
<th>Mutation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip</td>
<td>Removes an action from a trace</td>
</tr>
<tr>
<td>Repeat</td>
<td>Repeats an action from the trace to the trace</td>
</tr>
<tr>
<td>Swap</td>
<td>Swaps two (sub-)terms in the trace</td>
</tr>
<tr>
<td>Generate</td>
<td>Replaces a term by a random one</td>
</tr>
<tr>
<td>Replace-Match</td>
<td>Replaces a (sub-)term by another (sub-)term in the trace</td>
</tr>
<tr>
<td>Replace-Reuse</td>
<td>Replaces a (sub-)term by one of its sub-terms</td>
</tr>
<tr>
<td>Remove-and-Lift</td>
<td>Swaps an action of a different type</td>
</tr>
</tbody>
</table>

TABLE 1: Mutations

**Example 5.** Suppose we have an initial trace $A = \text{in}(s, ch) \cdot \text{out}(s, x)$ where $ch = \text{CHello}(cs, ext)$ 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 $ch_1 := \text{Remove-and-Lift}(ch)$
- add a cipher suite $c$ by $ch_2 := \text{Replace-Reuse}(ch_1)$

$$
= \text{CHello}(\text{sid}, [c, c, \text{nill}], [e_1, \ldots, e_{n-1}, \text{nill}]).
$$
• iterate Replace-Reuse $k$ times and obtain $ch_k := CHello(sid,[c,\cdots,c,nil],[e_1,\cdots,e_{n-1},nil])$
where $c$ is repeated $k$ times.

Sending $ch_k$ to the server triggers a HelloRetry-Re-
quest message due to the missing extension $e_n$, i.e.
when removing the supportedGroups extension. Then ap-
plying the Repeat mutation, we obtain the trace $A' =
in(s, ch_k).in(s, ch_s).out(s, x)$. We shall see in Section 6 that
additional mutations, notably involving cryptographic oper-
ators, 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 88) 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 [5, 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 looking at the heap state. Using
known or code instrumentation techniques such as ASAN [69],
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 tls puffin,
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: $S \to P(C)$. 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 pro-
PERTY 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 tls puffin Fuzzer

We present **puffin** and its derivatives, which altogether
is a public FLOSS framework [24] 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 (tls puffin) using puffin. We used tls puffin to fuzz
OpenSSL, LibreSSL, and wolfSSL. To show the modularity
and generality of puffin, we also briefly mention ss hpuffin,
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
tls puffin, and 2k for ss hpuffin.

#### 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. tls puffin for TLS and the 3 aforementioned
PUTs), puffin implements a DY fuzzer for them.

puffin builds on the state-of-the-art LibAFL [40] 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 [5, D.1]). We use the standard AFL-like code
edge coverage map [40] (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 represen-
tation of a trace, or compute the packets agents send in
a trace (using [8] and the Mapper). We implemented all
those features, which ease bug triaging (see Section 5.2).

tls puffin 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). 

tlsphinx 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.

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

### 5.2. Implementation Challenges and Features

We review some challenges we tackled in building pufin and tlsphinx, which are detailed further in [5, 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 [C.T.A] (how to concretize), DY traces, claims, etc. See [5, C.T.A] for a performance evaluation.

**Gathering Knowledge from Protocol Outputs.** Let us recall that when executing an action \( \text{out}(c, x) \), the message \( m \in M \) from \( \langle s'_c, m \rangle = \text{output}(s_c) \) gets added to the attacker’s state \( \phi \) by assigning \( m \) to the variable \( x \) (see Section 3.2). Moreover, the message \( m \in M \) is a bitstring, not a term \( t \in T \). 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 pufin 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 \( \phi \). 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 attacker’s state and its computation will now fail. We alleviate this issue with queries. When adding some (sub-)message \( m \) to \( \phi \), pufin and tlsphinx also stores from which agent \( c \in C \) 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 \( \phi \). 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. tlsphinx also offer features, notably its DSL, to allow developers to analyze TLS libraries with a similar goal of TLS-Attacker [42].

### 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 [74].

#### 6.1. Finding DY Attacks

Our first goal is to answer the following **Research Question:** How does tlsphinx 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 [46] 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 [32] 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 tlsphinx: 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 [64], StateAFL [61], and AFLnwe [6], which were already plugged into the ProFuzzBench [62] benchmark tool. All our experiments were conducted on

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.
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. 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 WolflSSL 5.3.0, which was affected by 6 vulnerabilities tlspuffin found; none of those 9 campaigns found any of them.

Moreover, OpenSSL and wolfSSL are well-tested software and fuzzers are already continuously testing them. In particular, Google’s large-scale oss-fuzz project continuously runs AFL++, honggfuzz, and libfuzzier [39] [54] [45] on OpenSSL as well as honggfuzz and libfuzzier on wolfSSL. The wolfSSL project itself runs 7 fuzzers internally every night (including a network fuzzer, libfuzzier, tlspuffin, and AFL) [79]. 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 [5] [C1].

**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.

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.

### 6.1.4. Summary of Vulnerability Descriptions (more in Section A.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 [66] 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 [5] [C5] why such a trace triggers the bug, its root causes, and how we proceed 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 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 bit-level fuzzers and others’ fuzzing efforts

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.

<table>
<thead>
<tr>
<th>CVE ID</th>
<th>AKA</th>
<th>CVSS</th>
<th>Type</th>
<th>New</th>
<th>Version</th>
<th>TLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2021-3449</td>
<td>SDOS1</td>
<td>5.9</td>
<td>Server DoS, M</td>
<td>✓</td>
<td>1.1.1j</td>
<td>1.2</td>
</tr>
<tr>
<td>2022-25638</td>
<td>SIG</td>
<td>6.5</td>
<td>Auth. Bypass, P</td>
<td>✓</td>
<td>5.1.0</td>
<td>1.3</td>
</tr>
<tr>
<td>2022-25640</td>
<td>SKIP</td>
<td>7.5</td>
<td>Auth. Bypass, P</td>
<td>✓</td>
<td>5.1.0</td>
<td>1.3</td>
</tr>
<tr>
<td>2022-38152</td>
<td>SDOS2</td>
<td>7.5</td>
<td>Server DoS, M</td>
<td>✓</td>
<td>5.4.0</td>
<td>1.3</td>
</tr>
<tr>
<td>2022-38155</td>
<td>CDOS</td>
<td>5.9</td>
<td>Client DoS, M</td>
<td>✓</td>
<td>5.3.0</td>
<td>1.2</td>
</tr>
<tr>
<td>2022-39173</td>
<td>BUF</td>
<td>7.5</td>
<td>Server DoS, M</td>
<td>✓</td>
<td>5.5.0</td>
<td>1.3</td>
</tr>
<tr>
<td>2022-42905</td>
<td>HEAP</td>
<td>9.1</td>
<td>Info. Leak, M</td>
<td>✓</td>
<td>5.5.0</td>
<td>1.3</td>
</tr>
</tbody>
</table>
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 [39] [81] [54] 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 $senc(sdec(x, k), k')$, which expresses that the attacker, instead of forwarding $x$, decrypts and re-encrpts $x$ 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 $[t']_a$ is very unlikely to be reached by bit-level mutations from $[t]_a$. 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).

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.

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 [66]; 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, [66] 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 [15].

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. [57]: “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 [55]. 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 $10^9$.) 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 [71] 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 6.3). 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 A.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 (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 $\Sigma$ (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 wolf/SSL 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.
ware Analysis Platform”. In: Communications of the ACM 64:8 (2021).


Finally, program verification of existing code, for instance in C, has been made possible with tools like Frama-C [14] combined with security proofs in tools like EasyCrypt [11], and secure compilers like CompCert [53]. However, this approach does not currently scale beyond simple cryptographic primitives, excluding cryptographic protocol implementations.

A.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 [66]. 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. tlsbuffin 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. tlsbuffin 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 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 [5] 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.

Appendix A.

Additional Content

A.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* [73] requires a dedicated protocol implementation and an effort. As an illustration, proofs for the record layer of QUIC [36] 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 [35] and for TLS 1.2 the proof (using F7) does not cover the full handshake [19]. For particular cryptographic primitives, there have been efforts to provide end-to-end proofs of highly optimized implementations 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* [82], 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* [16] 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 [9] 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.


A.3. Coverage Evaluation (continuing Section 6.2)

A.3.1. Part I: Comparing coverage of tlspuffin vs. bit-level fuzzers. We compared the coverage of tlspuffin against AFLnwe [3], AFLNet [64], and StateAFL [61], which are all based on the bit-level AFL fuzzer [39]. 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. We provide more insights in [3, 4].

A.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.

tlspuffin currently unearths 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 proves 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.


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 [3, 4].

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

<table>
<thead>
<tr>
<th>OpenSSL</th>
<th>tlsuffin</th>
<th>TLS-Anvil</th>
<th>wolfSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_p</td>
<td>25.8%</td>
<td>19.9%</td>
<td>29.0%</td>
</tr>
<tr>
<td>l_a</td>
<td>28074</td>
<td>21664</td>
<td>23421</td>
</tr>
<tr>
<td>b_p</td>
<td>20.2%</td>
<td>16.0%</td>
<td>19.9%</td>
</tr>
<tr>
<td>b_a</td>
<td>12363</td>
<td>9806</td>
<td>9434</td>
</tr>
<tr>
<td>f_p</td>
<td>30.1%</td>
<td>23.9%</td>
<td>31.9%</td>
</tr>
<tr>
<td>f_a</td>
<td>2370</td>
<td>1885</td>
<td>1225</td>
</tr>
</tbody>
</table>

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