

DDYF: Differential Dolev-Yao Fuzzing of Cryptographic Protocols*

v2.0 — October 14, 2026†

Tom Gouville
Université de Lorraine, CNRS, Inria, LORIA
Nancy, France

Lucca Hirschi
Université de Lorraine, CNRS, Inria, LORIA
Nancy, France

Steve Kremer
Université de Lorraine, CNRS, Inria, LORIA
Nancy, France

***Abstract*—Symbolic formal verification of cryptographic protocols based on the Dolev-Yao (DY) attacker model—an active attacker with full network control and perfect cryptography—is well-established for finding design-level logical flaws in cryptographic protocols. Building on this, Dolev-Yao Fuzzing (DYF) enriches fuzzing with this attacker model to uncover logical bugs at the implementation level. In contrast to bit-level fuzzers (*e.g.*, AFL), DYF leverages a formal model of messages and cryptography to generate structured, adversarial executions, such as replaying and re-signing a modified payload.**

However, a significant limitation of DYF is the requirement to precisely model properties to check at runtime (*e.g.*, session parameter agreement). Defining these properties is labor-intensive and inherently non-exhaustive, often necessitating complex instrumentation of the Programs Under Test. Consequently, typically only a subset of logical attacks is detected.

We address this limitation by introducing Differential Dolev-Yao Fuzzing (DDYF) based on a differential oracle to compare executions across different protocol implementations. By interpreting discrepancies through the DY model, it identifies semantic differences indicative of bugs or vulnerabilities, effectively minimizing false positives.

We propose a generic design for DDYF, implement it within the PUFFIN DY fuzzer, and evaluate it on four major TLS implementations and two SSH implementations. Our results demonstrate that DDYF can detect vulnerabilities that evade state-of-the-art fuzzers, specifically those requiring DY attacker capabilities (missed by bit-level differential fuzzers) or complex objective oracles (missed by DYF). DDYF also uncovered numerous new RFC violations in OPENSSL, BORINGSSL, WOLFSSL, LIBRESSL, LIBSSH, and WOLFSSH, which are by-design hardly detectable with non-differential oracles.

1. Introduction

Cryptographic protocols secure message exchanges in our everyday life. Typical examples are TLS, underlying HTTPS, EMV, securing credit card payments, or Signal, underlying the eponymous messaging app and WhatsApp. These protocols use cryptographic primitives, *e.g.* encryption and signatures, to guarantee security properties, *e.g.* confidentiality and authentication. By nature, such protocols are hard to design and implement correctly: they are often both stateful and concurrent and have to resist complex threat models. It is therefore unsurprising that they have been subject to numerous attacks both at the design, *e.g.* [14, 4, 27, 20, 19, 11, 12] for TLS, and implementation level, *e.g.* the Heartbleed [21] and Apple Goto fail [22] bugs among many other CVEs.

Formal Verification. Formal, symbolic verification, relying on tools such as PROVERIF [15] or TAMARIN [6], is an extremely successful approach for analyzing and finding vulnerabilities in protocol designs. For instance, Basin *et al.* [7] discovered a PIN bypass attack in EMV, and Bhargavan *et al.* [13] identified several flaws in Signal's post-quantum protocol. As documented in [39] the standardization process of TLS 1.3 has also been helped by intensive, continuous verification and attack finding efforts [20, 11, 19, 12]. This approach can be traced back to the seminal paper of Dolev and Yao [26]. It considers a powerful attacker that entirely controls the network and actively participates in protocols. In this model, messages are represented as formal terms and cryptography is idealized. For example the encryption of a message m with a key k is represented by a formal term $enc(m, k)$. Typically, the only possible manipulation of this term is to apply decryption with the key k, allowing one to recover m. This abstraction allows tackling large, comprehensive protocol models, and at least partially, automate the formal analysis. However, while this approach was designed to find *logical attacks* on protocol *specifications*, *i.e.* attacks that exploit flaws in the *protocol logic* such as downgrade, replay, authentication bypass attacks, it cannot analyze actual implementations. In particular, logical attacks that arise due to implementation bugs [9, 23, 30, 16] are out of scope.

Formally proving implementations does however require a tremendous amount of work and does not scale easily. Some works [24, 25] require a re-implementation in a proof-oriented language, and do not apply to existing, deployed implementations, while others require substantial code

* *This is an extended version of [31].*
† *A list of changes since the initial version can be found in Appendix G.*

annotation or proofs (proof-to-code ratio of 3:1 for [3]).

DY Fuzzing. Although *fuzzing* does not provide formal guarantees, it is an effective method for finding bugs and vulnerabilities at large scale. However, fuzzing cryptographic protocols is hard and most state-of-the-art techniques fail to capture the capabilities of a Dolev-Yao (DY) attacker [2]. To close this gap, the paradigm of Dolev-Yao Fuzzing (DYF) [2] and the PUFFIN fuzzer have been recently introduced. DYF transposes ideas from symbolic protocol verification to fuzzing: test cases are *symbolic traces* where messages are represented as formal terms, mutations on these traces correspond to possible actions of a DY attacker, and security properties such as authentication are checked by the so-called Objective Oracle, in addition to memory safety checks.

By design, DYF [2] is capable of producing test cases expressing all logical attacks of a DY attacker and executing them directly on protocol implementations. This does not mean that all such attacks will be detected in practice, as this depends on the capabilities of the Objective Oracle. In practice, this oracle only detects crashes and memory corruption errors as well as violations of pre-defined, formally specified security properties. For instance, Ammann *et al.* [2] fuzzed TLS stacks while only checking bilateral agreement between client and server, thereby missing a large spectrum of logical attacks such as downgrade, confidentiality breaches, and *RFC violations*, *i.e.* non-conformance bugs with respect to the protocol specification (RFCs). These pre-defined properties are also costly to specify and check, often requiring complex instrumentation of Program Under Test (PUT) and will always be limited in scope. Moreover, some properties such as functional correctness with respect to the specification are extremely hard to formalize, which means that most RFC violations will be missed in practice. Most other protocol fuzzers rely on even simpler oracles (*e.g.* crash or simple message checks) that miss even more logical flaws and DY attacks; we expand on this in Section 6.

We propose Differential Dolev-Yao Fuzzing (DDYF) to remedy this problem. The crux of our approach is a new Objective Oracle based on *differential testing* that identifies semantic behavior differences between protocol implementations by re-interpreting execution discrepancies through the DY attacker model. After triaging, a difference may indicate a bug or vulnerability in one of the implementations. By design, differential testing only detects bugs on which the PUTs behave differently, and thus misses flaws shared by all tested stacks. Such implementation bugs are unlikely to affect several independent stacks simultaneously, while flaws of the specification itself are out of scope and already addressed by formal DY verification.

Challenges. Integrating differential testing into DYF is not straightforward and raises several key challenges.

A first challenge is a *meaningful* comparison of protocol executions given the *inherent non-determinism* of cryptographic protocols (random values, nonces, *etc.*). For this, we leverage the DY model to partially re-interpret PUTs outputs, *i.e.* bitstrings, into structured terms which allow us to compare *(i)* the message structure, and *(ii)* message contents while selectively abstracting the randomized fields that we automatically identify.

A second challenge is comparison of *encrypted messages*. We extract necessary key material from PUTs, and building again on the underlying DY model, specify how to decrypt messages before structural comparison of the plaintexts, as for other messages.

The third challenge is related to *specification permissiveness*: a protocol specification may have different valid interpretations. A simple example is the serialization of sets, which often allow for different orders. A more involved example arises in the TLS RFC, which allows a server to either send a list of separate encrypted messages or coalesce these messages into a single encryption. Similar challenges arise for other protocols, such as SSH. In all these cases, the PUTs produce completely different bitstrings.

Fourth, DDYF can produce numerous bug candidates, as many as 68M for a 24h campaign. Deduplication and triaging of these differences are both crucial and challenging.

Contributions

Design. At the core of our new fuzzer design is our *differential Objective Oracle* that considers multiple comparison dimensions: execution status, content of exchanged messages, PUT internal state, as well as PUT error messages. As mentioned above, the main challenge is to define comparison functions for these dimensions that are able to identify *semantically meaningful* differences. The key insight is to leverage the underlying DY model to define these, which allows us to structurally interpret the behavior discrepancies and effectively minimize false positives, which would otherwise defeat the whole approach as practically all trace executions would differ in some way. In particular, we propose DY model-based abstractions to compare protocol messages containing random values, decrypt messages for comparison, and accommodate different valid protocol interpretations. We also adapt the DYF architecture to support execution of multiple PUTs, accessing internal state of PUTs for comparison, as well as key material for decryption. Finally, we leverage again the DY model to recognize semantically equivalent behaviors and effectively *deduplicate* them in the Objective Oracle, achieving 95% reduction of traces. Combining this at *triaging time* with structural information about the attacker's behaviors producing each of the remaining differences, we show how to effectively cluster them into a manageable number of *buckets* of similar bug candidates.

Implementation. We implement our DDYF design in the PUFFIN DYF library[1] [2]. We leverage PUFFIN's modular architecture such that our implementation of the DDYF logic, and most of our coding effort, is protocol-agnostic, coded once and for all. This effort augments past and future protocol-specializations of PUFFIN with DDYF capabilities.

1. PUFFIN [2] is based on a modular architecture with a protocol-agnostic kernel and specific parts that can be specialized for a particular protocol yielding a DY fuzzer for that protocol.

Evaluation. We chose to thoroughly evaluate our implementation of DDYF on TLS 1.2 and TLS 1.3. These two protocols—1.3 being a major re-design—are maybe the most widely used cryptographic protocols, have been subject to intensive fuzzing efforts, and are overly complex with intricate state machines. Hence, they are a good benchmark for our approach. For similar reasons, DYF was also evaluated on these protocols, easing comparison with DDYF. We tested widely used implementations of these protocols: OPENSSL, BORINGSSL, WOLFSSL, and LIBRESSL, always comparing OPENSSL with another implementation.[2] For each resulting pair, we triaged the bug candidates using our triaging methodology into buckets with unique root causes, *e.g.* for WOLFSSL, ca 4M traces into 145 buckets. Our triaging methodology is fully tool-supported with some aspects that remain manual. After having conducted a first triaging this way on OPENSSL vs WOLFSSL, we replicated our triaging process on other pairs using LLMs (augmented by our triaging tool support) that produce final artifacts that are concise enough to be manually verified. Finally, to show that DDYF is not tied to TLS, we also applied our fuzzer to SSH, testing two widely used implementations: LIBSSH and WOLFSSH.

Our evaluation demonstrates that: *(i)* DDYF finds all vulnerabilities of a carefully curated benchmark of known CVEs and is the first fuzzer to find two of them. *(ii)* DDYF uncovers new security-related bugs that prompted upstream fixes by the vendors upon our responsible disclosure. First, DDYF flagged behavioral differences that turned out to reflect four distinct postures toward the client-side defense (RFC 5746) against the CVE-2009-3555 renegotiation attack: OPENSSL enforces it by default (since 2021), LIBRESSL disables it through an option that is set by default, WOLFSSL compiles it only under an opt-in hardening flag, and BORINGSSL hardwires it off entirely. LIBRESSL additionally exposes a genuine "configuration trap": the option that turns the check off is set by default and by the macro most applications enable expecting safe hardening, which here instead keeps the defense off. We stress that because the weakened handshake still completes successfully, non-differential Objective Oracles have nothing to flag; only the divergence across implementations surfaces it. Second, DDYF found that LIBRESSL's Elliptic Curve implementation lacks a second line of defense, computing with the private key on an adversarial degenerate point before rejecting it. After our reports, both were addressed upstream: LIBRESSL and WOLFSSL now enforce the renegotiation defense by default, BORINGSSL restored strict enforcement as an opt-in, and LIBRESSL added the early curve rejection. *(iii)* In total, DDYF discovered 54 new acknowledged RFC violations in TLS and SSH stacks (49 and 5, respectively). Note that specification compliance cannot be checked in general by standard Objective Oracles while DDYF turns out to be particularly well suited for finding such bugs.

2. Using a single stack as the common baseline is a purely practical choice that limits the number of pairwise campaigns and the triaging effort. In principle, this is without loss of generality: two stacks that behave differently cannot both agree with the baseline.

We made DDYF and PUFFIN modular, which is crucial for reusability. For instance, our instantiation on SSH for LIBSSH and WOLFSSH reused the entire DDYF architecture as-is, which allowed us to eliminate the false positives and surface genuine, new, and acknowledged RFC violations.

Outline. We present some background in Section 2, and then discuss key challenges for DDYF in Section 3. Next, we specify our DDYF design in Section 4 and its implementation and evaluation in Section 5. Related work is discussed in Section 6 and Section 7 concludes.

Artifact. Our implementation and fully reproducible evaluation are available in our artifact [32].

2. Background

We combine differential and Dolev-Yao Fuzzing (DYF) to detect cryptographic protocol bugs; this section provides background on these topics.

2.1. Cryptographic Protocols and TLS

Cryptographic protocols are hard to test: executions consist of multiple round-trips, use complex message formats involving cryptography, and evolve internal state. A prominent example of such a complex cryptographic protocol is TLS [42], which we use as a running example and case study for our evaluation.

TLS is a widely adopted protocol that guarantees confidentiality, integrity, and peer authentication between a client and a server over an untrusted network. While the *handshake protocol* is responsible for setting up the secure channel, negotiating algorithms and authenticating the peers, the *record protocol* is used to securely transport application data after the handshake. We focus on the handshake protocol of the latest TLS 1.3 version, which is the most complicated part of the protocol. It negotiates cipher suites for the record layer, and establishes a master secret through an (Elliptic Curve) Diffie-Hellman (DH) key exchange and/or a Pre-Shared Key (PSK), *i.e.* a symmetric key established in previous sessions. For the sake of agility, TLS offers different modes that can be combined, resulting in a complex state machine [42, App. A].

A typical TLS 1.3 handshake (cf Figure 1 where we omit legacy components) starts with the client sending a `ClientHello` message advertising its supported cipher suites and signature algorithms, and either an ephemeral (EC)DH key share or a list of PSK identifiers (or both). If the server does not support the client's key share group, it may send a `HelloRetryRequest` containing the list of the server's supported groups to request a new key share in one of these groups. Otherwise, the server responds with a succession of five messages. First it sends `ServerHello` providing its own key share and the chosen PSK identifier if any. Using the derived session key and possibly the PSK, all remaining messages will be encrypted and integrity-protected (depicted by 🔐) with a MAC over the hash of the current transcript. Next, the server sends `EncryptedExtensions` (a list of optional

extensions not discussed here), `ServerCertificate` and `ServerCertificateVerify` messages, containing respectively its certificate chain and a signature of previous messages using the certificate's private key. The end of the server's message flight is indicated by a `ServerFinished` message containing a MAC over previous messages. Alternatively, the last four messages can be coalesced into a single encrypted message (depicted on the right).

After validating the server's identity and if the server requested client authentication in its `EncryptedExtensions`, the client sends its own `ClientCertificate` and `ClientCertificateVerify`. Finally, the client sends a `ClientFinished` message, providing key confirmation and also binding the participants' identities to the session key. Optionally, the server can then send a `NewSessionTicket` (not depicted) with cryptographic material to use for a future PSK-mode session. The record layer then secures application data using the negotiated cipher suite and the derived session key.

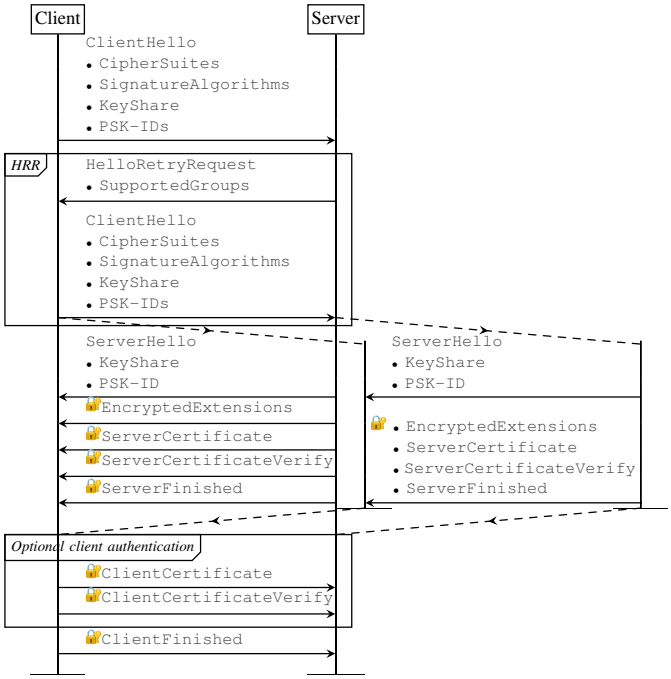

Figure 1: Some Typical TLS Handshakes

2.2. Fuzzing

Fuzzing is a well-established testing method for finding vulnerabilities and bugs in software [28, 48, 35, 37]. The general idea is to execute a program, called Program Under Test (PUT), on unexpected inputs until an Objective Oracle detects security policy violations, *e.g.*, a crash or memory corruption. These inputs are either mutated (*mutation-based*), starting from a set of initial inputs (Seeds), or generated based on a model of the input format such as a grammar (*model-based*). A Feedback loop guides the decision whether new inputs should be stored in an evolving Corpus of *interesting* test cases. In most cases, *code coverage* is used as feedback: a new test case is deemed interesting and added to the Corpus when exercising a new code path. An overview of the architecture of a typical mutation-based fuzzer is illustrated in Figure 2, using the terminology of LIBAFL [29].

Most fuzzers such as AFL, LIBAFL, LIBFUZZER are *bit-level* fuzzers, meaning that they operate on test cases which are *bitstrings* and apply mutations at the bit level (*e.g.* bit flips, byte increments). Their Objective Oracles are often limited to detecting crashes or memory corruption using sanitizers such as ASAN. In the spirit of property-based testing [18], some fuzzers [44] (including DYF introduced next) support more advanced policies such as checking runtime properties implemented as assertions or input validation [37].

In contrast, *differential fuzzing* uses different PUTs as cross-referencing oracles [37]. Each test case is executed on two or more implementations of the same program and their behaviors are compared to identify discrepancies hinting at bugs. This relies on the idea that it is extremely unlikely that two PUTs will behave wrongly —with respect to functional correctness for instance— in exactly the same way for the exact same input. This approach has the benefit of finding subtle differences without writing numerous complex security policies and effective ways to check them. It however requires the ability to *meaningfully* compare PUT executions.

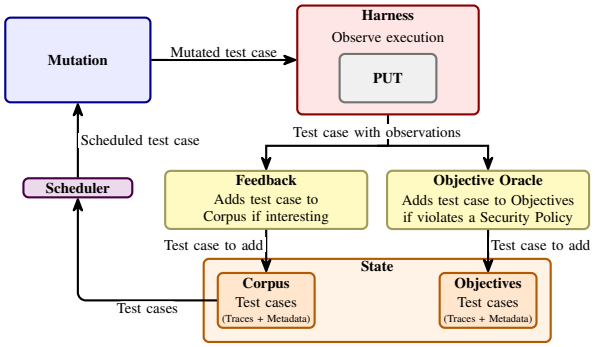

Figure 2: Fuzzer Architecture - Example of LibAFL [29]

2.3. Dolev-Yao Fuzzing

When testing cryptographic protocols, bit-level fuzzers have severe limitations that *DYF* [2] was designed to overcome.

2.3.1. Challenges with Cryptographic Protocols Fuzzing. Ammann *et al.* [2] show that bit-level fuzzers (including network fuzzers such as AFLNet [40], StateAFL [38], and AFLnwe [1]) and model-based fuzzers (including stateful [5] and state-machine learner fuzzers [41, 23, 30]) are ill-suited to find logical attacks in cryptographic protocols for at least one of the following reasons: *(i)* inability to handle

stateful and multi-round-trip protocols, *(ii)* failure to capture DY attacker capabilities and meaningful adversarial behaviors: *e.g.* input mutations or generations are not aware of cryptographic operations and are thus overwhelmingly likely to produce cryptographically invalid messages, *(iii)* protocol vulnerabilities such as an authentication violation are not detectable by standard oracles. Therefore, these fuzzers fail to reach bugs triggered by logical attacks, *i.e.* adversarial behaviors in the scope of the DY attacker model [26], and to detect protocol flaws caused by implementation bugs.

2.3.2. DY Fuzzing. This approach [2] was specifically designed to weaponize the DY attacker capabilities [26] against cryptographic protocols. It can be viewed as a DY attacker in a mutation-based fuzzing loop. The architecture of this fuzzer is similar to other mutation-based fuzzers as shown in Figure 2. The main difference lies in the notion of test cases which are structured DY execution traces (*DY traces*) instead of bitstrings. We next summarize the key concepts of DYF and refer the reader to [2] for more details.

Terms. DY models specify protocol messages and notably the ones computed by the DY attacker in a *term algebra* where cryptographic operations, formatting, and other message manipulations are modeled as operators over terms. For example, the term $\mathsf{senc}(m, k)$ represents the encryption of message m with the key k, where m and k are also terms. Terms can contain variables referring to message outputs by the PUT and received by the DY attacker.

Definition 1 (Terms [26, 2]). 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 set of *variables* $\mathcal{V}$. The set of terms $\mathcal{T}$ is defined inductively as the set containing atoms, $\mathcal{V}$, and terms obtained by applying operators to other terms.

Example 1. A basic model of a MAC, cryptographic hash function as well as a TLS `KeyShareEntry` (a structure containing the value of a key share and the group it belongs to) can be specified by a signature Σ that contains operators $\mathsf{MAC}(\cdot,\cdot)$, $\mathsf{checkMAC}(\cdot,\cdot)$, $\mathsf{H}(\cdot)$, and $\mathsf{KS}(\cdot,\cdot)$, and an atom $\mathsf{true} \in \Sigma$ (for valid MAC verification with $\mathsf{checkMAC}$).
Slightly simplifying, one could model the `ServerHello` message with an additional operator $\mathsf{SH}(\cdot,\cdot)$ and a term $sh_0 = \mathsf{SH}(\mathsf{KS}(k, g), p)$ where k models a key share in a group g and p is a PSK identifier, which require other operators not described here.

We now equip the term algebra with a semantics suitable for fuzzing that *interprets* terms as bitstrings $[\![\mathtt{u8}]\!]$. First, each operator $\mathsf{f} \in \Sigma$ (of arity i) is equipped with an *interpretation* $[\![\mathsf{f}]\!] : [\![\mathtt{u8}]\!]^i \rightharpoonup [\![\mathtt{u8}]\!]$. For instance, $[\![\mathsf{SH}]\!]$ implements the serialization of a `ServerHello` message given its components as bitstrings. Next, an *attacker state* ϕ maps each variable (representing a previous output) to a message: $\phi : \mathcal{V} \rightharpoonup [\![\mathtt{u8}]\!]$. Given ϕ, the interpretation of term t, denoted by $[\![t]\!]_\phi$, is then inductively defined: we simply apply $[\![\cdot]\!]$ on the operators and recursively interpret the subterms until reaching variables, which are looked up in ϕ. The component responsible for this is the Mapper.

Since operators are interpreted as functions over bitstrings, they naturally satisfy the expected algebraic properties (*e.g.* decryption cancels encryption). Hence, DYF does not require making these properties explicit through equational theories as symbolic models do; see [2, Remarks 1 and 2].

DY Traces. are sequences of network actions (input and output) a DY attacker can perform. The attacker communicates with agents identified by *channels* from $\mathcal{C}$, *e.g.* a channel c for a TLS client and s for a server. Simplifying from [2], an action is either an output $\mathsf{out}(c)$ or an input $\mathsf{in}(c, t)$ with $c \in \mathcal{C}$ and $t \in \mathcal{T}$ a term. The *output action* $\mathsf{out}(c)$ indicates that the DY attacker is waiting to receive a message from the agent on channel c, which will be parsed and stored in the attacker-state ϕ (see below). The *input action* $\mathsf{in}(c, t)$ indicates that the attacker computes a message $[\![t]\!]_\phi$ with the structure t and sends it to c, who *inputs* it. DY traces express the capabilities of a DY attacker [26]: forging, forwarding, dropping or modifying messages.

Example 2. Continuing Example 1, consider the trace $T = T_0.\mathsf{out}(c).\mathsf{in}(c, sh_0)$. After some prefix trace T_0, the DY attacker, here acting as an attacking server, waits for a message from client c, then sends a valid `ServerHello` message using term sh_0. One could imagine trying to test a TLS client implementation with a slightly different trace (obtained through *term mutations*) $T^m := T_0.\mathsf{out}(c).\mathsf{in}(c, sh^m)$ where $sh^m = \mathsf{SH}(\mathsf{KS}(k, g'), p)$ uses the wrong group in the key share entry to see how the client reacts. As we shall see, a similar trace triggers an RFC violation on WOLFSSL (Section 5.3.2).

Parsing and Knowledges. While mapping bitstrings to terms is not always possible (how to re-interpret the hash of some structured data?), one can *partially* parse protocol messages until reaching some *opaque* data such as ciphertexts, hashes, or signatures, *i.e.* data whose arguments cannot be recovered without additional information (*e.g.* a key). We thus assume a *partial parsing function* $\lfloor\cdot\rfloor : [\![\mathtt{u8}]\!] \rightharpoonup (\mathcal{T}, \Phi)$ that extracts as much structure as possible from a bitstring and returns a term t along with an attacker state ϕ mapping the variables of t to bitstrings, which cannot be parsed further. This corresponds to the knowledge extractions described in [2, §5.2]. Note that for a purely plain text message t without variables, one has $\lfloor [\![t]\!]_\emptyset \rfloor = (t, \emptyset)$.

Each time an output is executed and a new message m is received from the PUT, it is automatically parsed using $\lfloor\cdot\rfloor$ yielding some (t, ϕ). All sub-terms of t, including t, are then interpreted as bitstrings and stored in the attacker-state ϕ, associated with fresh variables. We call these *extracted* messages Knowledge. They can then be queried and used in input terms to be sent to the PUT.

Trace Mutations. As with other mutation-based fuzzers, DYF starts from an initial corpus of Seeds: here, traces written in a Domain Specific Language (DSL) that

express successful protocol flows ("happy flows" as per the specification). It then relies on two levels of trace mutations. Action-level mutations introduce changes in the series of input and output actions (removing, duplicating steps in the trace). Term-level mutations modify terms in input actions, *e.g.*, replace a sub-term by another (such as in Example 2), modify an operator in a sub-term, or even generate new terms from scratch. These mutations are designed to explore the space of DY attacker behaviors.

Objective Oracle. The objective oracle relies on Claims: data collected from the PUT's internal state through instrumentation, typically using callbacks of the PUT API. For instance for TLS, claims are triggered by peers when reaching the end of the handshake and contain their respective views on the session. The oracle then checks pre-defined security policies expressed as relations over these claims such as *authentication*: *i.e.* the finished claims agree on the handshake parameters (identities, cipher suite, transcript, *etc.*).

3. Overview and challenges

Our goal is to add differential fuzzing capabilities to DYF. The main idea is to execute the same DY trace on two different PUTs implementing the same protocol and to check for discrepancies between the resulting observations—such as output messages (Knowledges) or execution status—of both executions.

3.1. Overview

Differential Dolev-Yao Fuzzing (DDYF) requires several architectural modifications. First, the Harness needs to execute and observe two PUT instances in parallel. Two so-called Observers collect the execution status (*i.e.* success or error, and the last successfully executed trace action), the Knowledges (parsed output messages), and the Claims (some PUT state information) for both PUT executions. Second, the Feedback now takes two observations as input: we naturally use the union of the code coverage of both PUTs as the feedback metric, and leave the exploration of finer-grained differential coverage metrics for future work. Last but not least, the Objective Oracle is replaced with a Differential Oracle that analyzes observation differences and flags objective traces.

The different types of observations being compared conceptually allow capturing different families of bugs and vulnerabilities. Comparing the execution status and last executed action allows detecting when one PUT stops before the other, likely because it rejected a message, while the other PUT accepted it and progressed further in the trace. This is typically the case with denial-of-service and authentication-bypass. Comparing Knowledge (message content) allows detecting a parameter mismatch between PUTs, which is typical *e.g.* in confidentiality attacks (weak secret key) or in downgrade attacks since both PUTs end up negotiating different parameters such as cipher suites. We back up this reasoning with concrete examples in Section 5.3. Finally, we discuss the Claims in Section 3.1.2.

While conceptually simple, comparing such cryptographic protocol executions is not straightforward and raises several challenges.

3.1.1. Dealing with Non-determinism. Cryptographic protocols inherently produce non-deterministic outputs due to the use of random values that further propagate to whole messages through cryptographic operations. In most cryptographic protocols, all exchanged messages contain randomness to transmit cryptographic material (*e.g.* key shares) or to ensure security properties such as freshness or unguessability (use of nonces). *E.g.* all TLS 1.3 messages are such, even for the first message (`ClientHello`). Comparing the exchanged messages (as bitstrings) would thus lead to systematic false positives. We stress that using a uniform seed for PRNG would be insufficient. Indeed, different PUTs usually greatly differ in how random values are derived, since this is typically not specified by protocol standards.

Of course, differences between randomized values are expected and should not be considered as an objective. In order to avoid such false positives, our approach must be able to *recognize and ignore* differences specifically due to random values while still comparing the rest of the messages. As we shall see, re-interpretation of messages into terms in the term algebra is a key enabler to achieve this.

3.1.2. Dealing with Encryption. Most cryptographic protocols use encryption to protect the confidentiality of some messages (which are also randomized). Comparing ciphertexts would again lead to systematic false positives. *E.g.* in TLS 1.3, only the first `ClientHello` message and the first part (`ServerHello`) of the server's response are sent in clear; other messages are encrypted. Simply ignoring encrypted messages would miss most of the handshake. To overcome this, our approach must be able to access the plaintexts of all ciphertexts and compare their contents.

3.1.3. Dealing with Semantically Equivalent Syntactical Differences. A possibly more surprising challenge is the permissiveness of the protocol specification; *i.e.* there might be multiple valid and specification-compliant ways to perform a given action. For instance, different implementations may send a different flow of messages (or messages with different contents) to perform a semantically equivalent action according to the specification. Again, syntactically comparing the observations would lead to false positives, as it misses such semantic equivalences. An example for TLS 1.3 is the alternative flow of the 4 server messages that can be coalesced into a single encrypted message (depicted on the right of Figure 1). Both flows carry over the same information to the client, but the sent messages are entirely different. We encountered similar challenges with SSH.

This raises two challenges. First, when these syntactical differences are present in observations, our Differential Oracle must be able to *recognize* them as semantically equivalent and not flag them as objective. Another challenge

concerns the notion of test case (DY trace) itself. The terms in the input actions specify the computations performed by the DY attacker on the PUTs messages and must thus be agnostic to such syntactical message differences. *E.g.* continuing the TLS example and assuming the fuzzer acts as an attacking client: How can one specify a term for decrypting the server's messages that will behave equally on both flows? Otherwise, the execution would fail only on one of the two PUTs, systematically producing false positives.

3.1.4. Dealing with PUT configuration differences. Each PUT may have a different default configuration, *e.g.* some functionality might be enabled by default on only one PUT. In the case of parameter negotiations, PUTs may differ on cipher preferences. In the case of TLS (and similarly for SSH), each PUT has a list of supported ciphers and elliptic curves, a cipher preference order, enabled TLS extensions or capabilities, *etc.* It is thus important to find a way to align the PUT configurations to avoid systematic false positives due to such differences.

4. Differential Dolev-Yao Fuzzing Design

Our DDYF framework relies on the architecture and key concepts described in Section 3. In particular, the main challenge has been to design a new Differential Objective Oracle in place of the DY Objective Oracle (see Section 2.3). Crucially for this, we leverage the DY model underpinning DYF to solve the problems identified in Section 3. All the features presented in this section are nevertheless novel and specific to DDYF. We depict the overall architecture of this new oracle in Figure 3 and explain its components in Section 4.1. We also discuss how to deal with false positives stemming from real but benign differences (Section 4.2). Finally, we explain our triaging methodology and partially automated workflow to deduplicate, cluster, and investigate the objectives (Section 4.3). We mostly use TLS as a running example to illustrate the design of our Oracle, but the design is generic and can be instantiated for other protocols, as we show in Section 5.1 with SSH.

4.1. Differential Objective Oracle Design

Our differential oracle compares different dimensions that we detail in Sections 4.1.1 to 4.1.3. We also explain how we deal with semantically equivalent differences stemming from specification permissiveness (Section 4.1.5) and deduplication of objectives (Section 4.1.6).

4.1.1. Compare Execution Status. We consider two classes of errors: *PUT-specific errors*, sent to the application layer (*e.g.* with return codes) and *protocol-specific errors*, sent on the network and defined in the protocol specification.

Each implementation may use custom return codes and names for PUT-specific errors that are thus generally incomparable; they are nonetheless logged to ease the triaging (see Section 4.3). We also follow the PUT API documentation to divide them into those that are *fatal*, on which we stop

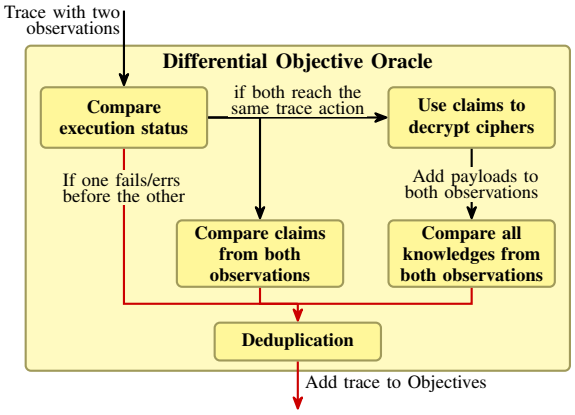

Figure 3: Differential Objective Oracle workflow

the trace execution, and the others that are considered as *warnings*, which we simply log. Accordingly, our oracle reduces the *trace execution status* to either Success (the trace was fully executed without a fatal error) or Error with the action where the trace execution stopped. The Observer collects such information through return codes. The Oracle compares both PUTs' execution status and flags a difference if one PUT stops before the other or if their statuses differ (*Error* vs *Success*). As we shall see (Section 5.3.2), some RFC violations we found correspond to cases where one PUT accepts an ill-formed message and progresses further while the other rejects it and stops with an error; we found 9 such *Wrong Accept* bugs (7 in TLS and 2 in SSH).

Protocol-specific errors are sent in protocol messages, as described in the protocol specification and must thus be equal across PUTs. For example in TLS, these errors correspond to `Alert` messages and are compared as part of the protocol messages (Section 4.1.2). Typically, when both PUTs stop at the same step with an Error status, they still may have sent different protocol-specific error messages; we found 15 such RFC violations.

4.1.2. Compare Messages through Knowledges. Recall that DY fuzzers partially parse the messages and store them as Knowledges. Both the whole exchanged message but also all the subcomponents that could be parsed are stored, once interpreted as bitstrings.

Example 3. A TLS `ServerHello` bitstring $m : [\mathsf{u8}]$ is parsed as $\lfloor m \rfloor = (sh, \phi)$ with $sh = \mathsf{SH}(\mathsf{KS}(x, y), z)$, $\phi = \{x \mapsto m_k, y \mapsto m_g, z \mapsto m_p\}$, where m_k is a key share, m_g a group, and m_p a list of PSK ids (this is simplified as these fields are structures themselves). The produced knowledges are m, m_k, m_g, m_p as well as $[\![\mathsf{KS}]\!](m_k, m_g)$, *i.e.*, the bitstring representing the key share term $\mathsf{KS}(x, y)$.

Filtering Knowledges. A straw man approach for comparing two observations is to compare each pair of elements in the two lists of knowledges. However, this would lead to systematic false positives for several reasons. First, as mentioned before, knowledges that contain randomness

will differ. Second, some knowledges may be legitimately lacking for one PUT. For example, a TLS implementation may support fewer certificate signature algorithms than another, resulting in fewer Knowledges. Instead, prior to comparison, we filter out all Knowledges that are expected to legitimately differ. This filtering is based on a protocol-dependent blacklist of Knowledge types. As we shall see in Section 5.1, this is implemented by simple code annotations. Importantly, we still produce blacklisted knowledges since they are crucial for preserving attacker capabilities: the attacker can *use them*, but the oracle does not *compare them*. For TLS, we blacklist all randomized fields (24 out of 137 TLS structure fields), yet our oracle is able to meaningfully compare messages containing such fields. Continuing Example 3, only m_k is blacklisted, meaning comparing m would result in comparing the term-structure sh of m as well as the bitstrings m_g and m_p but not m_k.

Comparing knowledges. After filtering the two lists, we flag the trace as objective if, for one pair of Knowledges, one of the following fails: *(i)* they were produced at the same trace action by the same agent, *(ii)* they have the same structure (and type) when interpreted as DY terms and recursively compare their sub-terms, *(iii)* their interpreted bitstring values are equal if they do not contain blacklisted fields. The DY abstraction of messages as terms is crucial for this. Comparing Knowledges enables detecting several types of bugs such as lacking or wrong protocol-specific error messages (*e.g.* we found 35 `Alert`-related bugs) or parameter negotiation issues.

Identify fields to blacklist. To identify fields to be blacklisted, we propose a simple yet efficient semi-automated methodology: we run a differential fuzzing campaign between a PUT and itself. All observed differences result from randomized values which can be quickly identified and tagged to be ignored. In practice, this method allowed us to easily identify all randomized values for TLS.

4.1.3. Compare PUT States through Claims. Recall that the fuzzer accesses the internal state of a PUT using Claims. Comparing Claims is similar to comparing Knowledges: filter, check types and compare the content. But there are some specificities: as Claims are produced through code instrumentation, callbacks provided by the PUT API, their behaviors do not perfectly align.

First, timing of Claims may differ. For example, the `ServerFinished` TLS Claim may be triggered when the server derived the session master secret, *i.e.*, after sending its first encrypted flight, or after receiving the `ClientFinished` message depending on the implementation. However, given a trace, each PUT should traverse the same sequence of states, though not necessarily at the same steps. Hence we omit the comparison of the action at which a Claim is produced.

Second, some Claims might be produced multiple times. For example, TLS `ServerFinished` claims may appear at every step after the handshake is finished, *i.e.*, when receiving `ClientFinished`, as well as when the server sends its encrypted session tickets. Since we are only interested in comparing the *state transitions* of the PUTs, we simply deduplicate equal successive Claims. Note that each trace is fully executed on both PUTs before their observations are compared, so the two executions need not be synchronized.

Finally, Claims among different agents may be reordered, *e.g.* a client claim may precede a server claim on one PUT, while another PUT may reverse them. We therefore compare Claims separately for each agent. This way of comparing Claims notably enables the detection of state-machine flaws, such as authentication bypasses.

4.1.4. Decrypt Ciphertexts. As explained in Section 4.1.2, encrypted messages must be blacklisted and not compared. However, this would imply ignoring most protocol messages. For instance, all TLS handshake messages following `ServerHello` are encrypted. Fortunately, the DYF framework offers the necessary ingredients to decrypt and meaningfully compare those messages. First, ciphertexts are stored as Knowledges and decryption keys, being part of the PUT internal state, can be accessed through Claims. Second, encryption and decryption are already operators in the DY model (see Definition 1) to enable the DY attacker to build terms and messages. We emphasize that the decryption keys obtained through Claims are only accessed by the Oracle for comparison, but not exposed to the attacker.

We still need to specify how exactly a given message should be decrypted. To do so, we leverage the DSL of DYF that offers human-readable and convenient ways to manually write terms (and traces). We write, once per protocol, a few *decryption recipes*, *i.e.*, terms that specify the functions and key material to apply on a given ciphertext. Our Oracle evaluates these recipes on Knowledges of ciphertext type, after the trace has been executed and before Knowledges are filtered and compared. If the evaluation of a *decryption recipe* for a given Knowledge succeeds, the resulting payload is treated like a new output message, *i.e.* we extract all possible Knowledges. An evaluation that fails is simply ignored as the trace may have finished before all necessary arguments of the recipe were collected. Failure to decrypt on one PUT only is detected during Knowledge comparison as Knowledges resulting from decryption are missing on the failing side.

For TLS, we wrote two decryption recipes, specifying how clients normally decrypt server messages (`EncryptedExtensions`, `Certificate`, *etc.*) and servers decrypt client messages (*e.g.* `ClientFinished`).

4.1.5. Dealing with Semantically Equivalent Differences. As already discussed in Section 3.1.3, specification permissiveness often yields semantically equivalent and yet syntactically different ways to produce or check messages. We already provided TLS examples witnessing this problem (coalesced flow and extensions ordering) and our strategy to solve these issues. First, we use the DY model and the term algebra to write new operators that implement the required semantic abstractions. Second, we sometimes write custom comparison routines for some protocol sub-message types that absorb benign syntactical differences (*e.g.* sets

are ordered in messages). We ease implementing the latter with lightweight code annotations that allow us to override default comparison functions. We have successfully applied this approach to TLS, and believe it is general enough to be applicable for other cryptographic protocols. We now provide more details on two concrete examples.

Extension order. In TLS, the order of extensions in the handshake messages is unspecified, which leads to false positives if two implementations send the same extensions in a different order. Therefore, we treat the extension comparison not as a list comparison but as a multiset comparison (to account for duplicate extensions).

TLS handshake variants. Recall that TLS supports two variants of the handshake protocol: the server either encrypts each message of its flight separately, or sends a single encryption of the list of these messages (see Figure 1).[3] In previous work on DYF [2], encryption was defined on a single message (of type `OpaqueMessage`), resulting in a single payload (of type `Message`). In particular, for the non-coalesced flow, the DY attacker could (through mutating an input term) access the `ServerCertificate` by decrypting the third server message stored in the Knowledges. The resulting trace distinguishes a PUT using a non-coalesced flow such as OPENSSL—for which this decryption will succeed—from one that uses a coalesced flow such as BORINGSSL, for which decryption fails since the third server message does not refer to the correct message. While the messages syntactically differ, they ultimately contain the same set of plaintext messages and should not be considered different.

We therefore introduced a new abstraction by adding operators to the term algebra: `MessageFlight` and `OpaqueMessageFlight`. When several encrypted messages are sent in the same agent action (in a so-called *flight*), *e.g.* the four encrypted messages following `ServerHello`, they are grouped into a single `OpaqueMessageFlight` message. We also introduce operators for encryption and decryption of such flights: $\mathsf{enc_f}(\cdot,\cdot)$ encrypts a `MessageFlight` of plaintexts yielding an `OpaqueMessageFlight`. Conversely, $\mathsf{dec_f}(\cdot,\cdot)$ decrypts an `OpaqueMessageFlight`. The `OpaqueMessageFlight` is either a single encryption of l plaintexts $p_1,\ldots,p_l$, or l ciphertexts $c_1,\ldots,c_l$ where c_i encrypts p_i. In both cases, applying $\mathsf{dec_f}(\cdot,\cdot)$ will transparently return the same result $\texttt{MessageFlight}(p_1,\ldots,p_l)$. This allows us to directly refer to message flights as four decrypted messages and have uniform traces that execute on all TLS implementations. Besides enabling comparison of PUTs that would otherwise produce systematic false positives, such abstractions turned out to be crucial to enable detecting a (known) CVE that only manifests when using the coalesced flow in an adversarial way.

4.1.6. Objective deduplication. While we aim to maximize the objective diversity, we also end up with a lot of similar traces producing identical differences that needlessly increase our triaging time. We thus introduced a deduplication mechanism that skips objectives with a set of differences and PUT output status identical to some prior objective originating from the same corpus trace. This approach is very conservative to avoid losing objective diversity but reduces the number of objectives by up to 95%, *e.g.* from 68M to less than 4M (see Appendix D).

3. To interoperate, stacks must accept both variants, but each outputs only its preferred one.

4.2. Dealing with Benign Differences

We discussed in Section 4.1 how our Oracle drastically reduces false positives. Nevertheless, some semantically different, yet legitimate, traces could still be flagged. Differing PUT configurations that are difficult to align are one such example. Since such differences are not bug candidates, they need to be identified and ignored as they would otherwise occur for nearly every trace, making triaging impossible. This requires particular care and fine-grained control to avoid introducing blind spots in our oracle and missing bugs.

PUT Configurations. An important source of differences is the PUT configuration: two TLS implementations may support different ciphers, curves or extensions, resulting in benign differences. We mitigate this by identifying a protocol-specific common set of parameters supported by both PUTs. Whenever possible, we rely on API-provided runtime, rather than compilation-time configuration, to use the default build configuration which is likely the most commonly deployed.

Other Differences. Even with aligned PUT configurations, some benign differences may remain. For some of them, blacklisting a specific field of a message when comparing Knowledges is not desirable as it may hide interesting behaviors. For example, we discovered that OPENSSL sends an encrypted `EllipticCurve` extension in `EncryptedExtensions` while WOLFSSL does not: in this case we neither want to blacklist `EllipticCurve` extensions altogether, nor ignore the content of `EncryptedExtensions` as we would miss differences on other extensions. We therefore use a protocol-specific *meta difference blacklist* that specifies specific contextual differences to ignore. We thus blacklisted differences of the form "`EllipticCurve` is missing in an `EncryptedExtension` message". Such blacklist rules need to be identified during triaging of initial campaigns and implemented as simple filtering rules on a pair of observations allowing the Differential Objective Oracle to filter out such benign differences.

4.3. Triaging and Analyzing Objectives

As is standard, *triaging* is crucial to assess the bug candidates, and yet may be time-consuming and tedious. This is exacerbated in differential fuzzing as numerous objectives mix benign differences or minor bugs with serious bugs. *E.g.* for TLS, DDYF 24h campaigns on 8 CPU cores for OPENSSL-WOLFSSL can produce ca 4M traces after deduplication (see Section 5.3.2).

We thus designed and implemented a three-step, semi-automated triaging methodology to deal with this high volume. The goal is to cluster all traces into a manageable number of *buckets*, each corresponding to a single *root cause*, which can be investigated. We end up with a *triaging script* that applies trace- and difference-based conditions on all objectives from a campaign to automatically cluster them into buckets, which are then labeled as a security-related bug, an RFC violation, a minor bug, or a benign difference. Once completed, this triaging script is reusable: it can be applied to a new campaign of the same stacks, possibly of different versions, to automatically identify the traces with new root causes that need further triaging. We summarize here the three triaging steps.

S1: Difference-based triaging. First, a script relying on PUFFIN[4] automatically produces the list of unique differences observed in the objective traces. We deduce from these a first list of buckets whose conditions are solely based on the observed difference (impacted agent, type of difference such as execution status, or Knowledge, PUT-specific error, *etc.*). On the above-mentioned fuzzing campaign the 4M objectives were triaged in about 50 buckets.

S2: Bucket refinement. This step identifies a single *root cause* for each bucket by iteratively splitting buckets into smaller and more constrained ones, until each bucket corresponds to a single behavior and isolated root cause. The bucket selection criteria now additionally depend on the objective trace, *i.e.* the adversarial behavior that triggers the difference such as the presence of an operator or uncommon extension at a specific trace action. As expected, this per-bucket work is more tedious, with investigation times that vary from a few minutes to several hours. To ease this process, we leverage our rich PUFFIN-based tooling to re-execute traces and observe differences, modify traces using the PUFFIN DSL and re-execute them, try to cleanly reproduce over TCP, *etc.* Continuing the previous example, we ended up with 137 buckets at this stage.

S3: Bucket assessment. Once triaged, the final step is the assessment of each bucket: does the root cause reveal a security-related bug, an RFC violation, a minor bug, or a benign difference to be ignored? Again, we heavily rely on the aforementioned tooling for this. We emphasize that this step is not specific to our approach or differential fuzzing: any bug candidate found by fuzzing needs assessment.

Cost and experimentation with LLMs. We first designed and implemented this triaging methodology on a TLS 1.3 OPENSSL-WOLFSSL campaign. Altogether, this took 2 person-months of work. Once we established a clear methodology specification and tooling, we experimented with the use of Large Language Models (LLMs), assisted by our tooling, to replicate triaging on other campaigns for TLS 1.2 and 1.3 together and the other stacks, as well as on SSH campaigns. The LLM is strictly confined to follow the methodology and to produce the triaging script, root-cause analysis, bucket assessment, and reproducers, which are verifiable and small enough (hundreds of LoC) to be reviewed by a human expert. This way provided a full triaging in a few days with a dozen rounds of human feedback, followed by a final review —all such results have been manually audited. This efficiency is not surprising given the structured nature of the task, which essentially consists in identifying patterns in the differences and traces (step S2) and following vendor code exercised by our objectives to identify root causes (step S3). LLM-free but tool-assisted triaging remains viable though. We evaluate the reliability and effectiveness of LLM-assisted triaging in Section 5.4.3.

4. For this, we developed a PUFFIN command to display all types of differences in both human- and machine-readable ways (like a `git diff`).

5. Implementation and Evaluation

5.1. Implementation

PUFFIN [2] is a generic FLOSS RUST library based on LIBAFL [29] for deriving DY fuzzers for arbitrary protocols. We implemented [32] our DDYF design (Section 4) in PUFFIN. We then tested it on four TLS stacks and two SSH stacks. Our DDYF implementation inherits PUFFIN's modular design and thus benefits both past and future specializations to other protocols and PUTs.

5.1.1. Generic DDYF in PUFFIN. DYF and PUFFIN are built around a three-layer, modular architecture. First, the fuzzing loop and components (mutations, generic term algebra and Mapper, *etc.*) are implemented in a protocol-agnostic way. Second, the protocol-layer implements protocol-specific components such as Seeds (written in a convenient DSL) and specializes the core components with protocol-specific information such as the operators, term interpretation and parsing for the Mapper, Observer, and Knowledge, *etc.* Finally, the PUT-layer implements an instance of the Harness for a given PUT. We follow this modular architecture to implement DDYF in PUFFIN. In particular, our Differential Objective Oracle is implemented once-and-for-all in a protocol-agnostic way and relies on the term algebra and its annotation interface provided by the protocol-layer explained next. Similarly, based on the existing components, we build the new Differential Harness and Differential Observer. All of these new components have a minimal interface with the protocol-layer to be specialized for each protocol. This, protocol-agnostic, once-and-for-all and reusable code represents ca 2.5 kLoC.

5.1.2. Adding New Protocols and PUTs. Integrating new protocols for DDYF fuzzing is in many respects similar to standard DYF, *e.g.* the same Mapper can be used. In contrast, DDYF fuzzing does not require writing protocol-specific security policies and ways to check them. For DDYF, the protocol-layer interface has been extended with the Knowledge code annotations including the blacklist of ignored fields and custom comparison functions. The interface also includes a set of parameters supported by all PUTs (and an API for aligning the PUTs configuration options), the *decryption recipes* (Section 4.1.4) and a new Claim for accessing encryption keys. We implemented

support for TLS, which represents ca 1 kLoC, with 2/3 being boilerplate and 1/3 being code annotations, decryption recipes and new PUT configuration APIs. We stress that TLS and SSH (which we also integrated, see below) are overly complex protocols, and that supporting other protocols would likely require less code.

Finally, we extended the existing four TLS PUT harnesses by around 200 new LoC for each to align their behaviors; to be compared with the existing (DYF) 1k to 1.5k LoC for each of them. For each of those PUTs, we support a range of versions, each additional version requiring 0 to less than a dozen lines of code. More details are given in Appendix A.

Genericity beyond TLS. To show DDYF is not tied to TLS, we then instantiated it on a second protocol, SSH, differentially fuzzing two independent server stacks (LIBSSH and WOLFSSH). This integration reused the entire DDYF architecture as-is, including the Differential Oracle and triage tooling. The integration took about one LLM-assisted person-week. Its DDYF-specific part (ca 700 LoC, less than 10% of the SSH protocol-layer for DYF) consists of annotating the fields to be ignored by the comparison (a dozen), aligning the configurations of the two PUTs, writing the decryption recipes and the Claim exposing the session keys (ca 150 LoC), and, for the remainder, aligning message flights and filtering the benign differences permitted by the SSH RFCs. Most of this code is boilerplate; what takes time and protocol expertise is the iteration between triaging the differences observed during fuzzing and refining the annotations, decryption recipes, and Claims until the abundant but benign objectives we initially observed are filtered out. Interestingly, every Differential Oracle feature we first introduced for TLS was also needed for SSH, and together they sufficed for the experiments that surfaced several RFC violations (Section 5.3.2).

5.2. Evaluation Research Questions

Once implemented, we evaluated our DDYF implementation to answer the following Research Questions:

- **RQ1**: Is DDYF capable of extending the detection capabilities of the existing DY Oracle and finding new types of bugs and vulnerabilities in real-world implementations?
- **RQ2**: Are the results produced by DDYF accurate and meaningful? What is the proportion of bugs versus benign differences found? What is the nature of the bugs found?
- **RQ3**: How do DDYF and DYF efficiency compare?
- **RQ4**: Are all components of our Differential Objective Oracle useful at finding bugs?

We neither evaluate the Feedback and achieved coverage nor the DY mutations, nor do we extensively compare PUFFIN with bit-level fuzzers, as these aspects are entirely orthogonal to our differential extension and have already been studied in-depth for DYF [2, §6].

5.3. DDYF Results

5.3.1. Detection of Known Vulnerabilities (RQ1).

Methodology. We first created a benchmark of existing, relevant CVEs. We collected all CVEs (filed until October 2025) affecting PUT versions that PUFFIN fully supports: OPENSSL 3.1.2 to 3.4.0 and WOLFSSL 5.0.0 to 5.8.0. We refined this list of 93 CVEs to those caused by DY attacker capabilities, *i.e.* logical attacks. While others are possibly in scope for DDYF, they also are for standard bit-level fuzzers. We further refined the list to those that require a differential or a DY oracle, excluding *e.g.* memory-related bugs, since they could as well be caught by memory sanitizers. Finally, we excluded CVEs that are totally out of scope of DYF or DDYF, *e.g.* side-channel attacks, weak cryptography or hardware-specific vulnerabilities.

The resulting benchmark (see Table 1) contains 5 CVEs affecting WOLFSSL corresponding to different types of logical vulnerabilities. More precisely, C1 is a certificate check bypass, C2 an authentication bypass, C5 is a downgrade attack in which a malicious TLS 1.2 server forces a TLS 1.3 client to accept a cipher suite it did not agree to, whereas C3 and C4 both rely on the PUT accepting incorrect messages. While the resulting benchmark is small, it covers various types of logical attacks. Moreover, there is an intrinsic reason for this: most known CVEs are memory corruption bugs, for which mature and diverse fuzzing techniques already exist, hence a natural bias. Incidentally, 2 of the 5 CVEs in our benchmark were first found by PUFFIN, two others were found by symbolic execution [47].

For each of those CVEs, we checked whether PUFFIN was able to find the vulnerability in a fuzzing campaign. For this, we wrote a specific triaging filter looking for the criteria of the vulnerability. (Of course in a real fuzzing campaign, this information would not be available beforehand but would surface from the triaging, see Section 4.3.) We then ran 50 differential fuzzing campaigns of 5h each on 8 CPU cores between the vulnerable WOLFSSL version and the latest OPENSSL version we support.

TABLE 1: Finding CVEs in 50 differential fuzzing campaigns running on 8 AMD EPYC 9275F CPU cores for 5h

Alias: CVE	DDYF	DYF	Prop.	Time To Find		
				Min.	Mean	Std. dev.
C1: 2022-25638	✔	✔	66%	59s	2328s	2724s
C2: 2022-25640	✔	✔	100%	<1s	38s	35s
C3: 2023-3724	✔	✘	100%	<1s	2s	1.2s
C4: 2023-6937	✔	✘	28%	420s	7130s	5151s
C5: 2024-5814	✔	✔	100%	<1s	8.5s	9.5s

Note: **DDYF** indicates that the CVE has been found using the DDYF extension of PUFFIN and **DYF** when found by PUFFIN's standard DY security oracle, **Prop.** is the proportion of campaigns where the CVE has been found, **Min.** is the minimum time to find the CVE across all campaigns, **Mean** the mean time, and **Std. dev.** the standard deviation.

Results. Table 1 shows the results of these fuzzing campaigns. All five CVEs were found. 3 CVEs could already be detected by DYF. Yet, in stark contrast to DYF, our approach does not rely on manually written security policies.

However, DYF failed to find[5] C3 and C4. Indeed, these two CVEs neither trigger memory corruption errors (captured by ASAN) nor violate authentication properties (captured by DYF security policies). Importantly, C3 causes a client to accept a `ServerHello` with a missing `KeyShare` and to use a weak encryption key. Detecting this *confidentiality attack* in a DYF framework was identified as a key problem for future work [2, §7]. Moreover, C4 uses a malformed coalesced `ServerHello` and extensions to make a client accept unencrypted server data. The client does not complete the handshake though and would thus not violate authentication properties or DYF security policies. And yet, both CVEs were found by our Differential Oracle. Note that C3 and C4 were first discovered using symbolic execution [47], which is orthogonal to fuzzing and relies on a manually written TLS specification. To the best of our knowledge, our DDYF extension of PUFFIN is the first fuzzer to be able to find these two CVEs and such logical attacks.

5.3.2. New Bugs and RFC Violations (RQ1,RQ2).

Methodology. To check whether PUFFIN could find new bugs and vulnerabilities, we ran three 24h TLS differential fuzzing campaigns on 8 AMD EPYC 9275F CPU cores, testing OPENSSL 3.4.0 against WOLFSSL 5.8.0, LIBRESSL 4.2.1, and BORINGSSL (May 2026)[6]. We then triaged the resulting differences for each of these campaigns using our tool-supported triaging and analysis methodology (Section 4.3), resulting in 145 buckets against WOLFSSL, 84 against LIBRESSL, and 154 against BORINGSSL.

Results. Out of those buckets, 49 correspond to unique RFC violations (see Section 5.4.1 for the other buckets), among which four are *Security-Related* bugs (two in LIBRESSL, one in WOLFSSL, and one in BORINGSSL). We list and reference all found bugs in Table 2 and summarize some findings here; more details can be found in the companion artifact [32] and Appendix E.

Security-Related RFC Violations. The four *Security-Related* bugs of Table 2 stem from two defense-in-depth gaps that DDYF surfaced—not independently exploitable vulnerabilities, but failures to uphold backstop protections that mature TLS stacks implement; both prompted upstream fixes following our reports.

The first is a missing client-side defense against renegotiation attacks (RFC 5746), which DDYF surfaced by comparing all four evaluated stacks (three bugs, one per affected stack). RFC 5746 introduced the `renegotiation_info` extension to counter the CVE-2009-3555 renegotiation/prefix-injection attack. Because the attack appears to the victim as a single initial handshake—only the server observes a renegotiation—the client's own defense is to require the extension already on the initial `ServerHello` (RFC 5746 §4.1). Only OPENSSL enforces this by default, having removed `SSL_OP_LEGACY_SERVER_CONNECT` from `SSL_OP_ALL` in OPENSSL 3.0 (in 2021). LIBRESSL and BORINGSSL, both OPENSSL forks, kept the pre-fork

PUT version	Agent	Type	Link
WOLFSSL 5.8.2	Client	Wrong Accept	#9240
WOLFSSL 5.8.2	Server	Wrong Accept	#9247
WOLFSSL 5.8.2	Client	Wrong Accept	#9331
WOLFSSL 5.8.2	Server	Wrong Accept	#9362
WOLFSSL 5.8.2	Client	Wrong Alert	#9503
WOLFSSL 5.8.4	Client	Security-Related	Fix
WOLFSSL 5.8.4	Server	Wrong Accept	#10016
WOLFSSL 5.8.4	Server	Wrong Alert	#9521
WOLFSSL 5.8.4	Client	Missing Alert	#9520
WOLFSSL 5.8.4	Client	Missing Alert	#9531
WOLFSSL 5.8.4	Client	Wrong Alert	#9639
WOLFSSL 5.8.4	Server	Wrong Alert	#9640
WOLFSSL 5.9.1	Client	Wrong Alert	#10746
WOLFSSL 5.9.1	Client	Wrong Alert	#10761
WOLFSSL 5.9.1	Client	Wrong Alert	#10792
WOLFSSL 5.9.1	Client	Duplicate Alert	#10746
WOLFSSL 5.9.1	Client	Missing Alert	#10791
OPENSSL 3.4.0	Client	Wrong Reject	#31533
OPENSSL 3.4.0	Client	Wrong Accept	#31531
OPENSSL 3.6.0	Client	Wrong Alert	#29353
OPENSSL 3.6.2	Client	Wrong Alert	#30818
BORINGSSL 0.20260327	Client	Security-Related	Fix #1 Fix #2
BORINGSSL 0.20260327	Server	Wrong Alert	#500638473
BORINGSSL 0.20260526	Server	Wrong Alert	#524759052
BORINGSSL 0.20260526	Client	Wrong Alert	#524759052
LIBRESSL 4.3.1	Client	Security-Related	Fix
LIBRESSL 4.3.1	Client	Security-Related	Fix #1 Fix #2
LIBRESSL 4.3.1	Server	Wrong Alert	#1279
LIBRESSL 4.3.1	Server	Missing Alert	#1279
LIBRESSL 4.3.1	Client	Missing Alert	#1279
LIBRESSL 4.3.1	Client	Missing Alert	#1279
LIBRESSL 4.3.1	Client	Missing Alert	#1279
LIBRESSL 4.3.1	Clt+Srv	Missing Alert	#1279
LIBRESSL 4.3.1	Client	Missing Alert	#1279
LIBRESSL 4.3.1	Client	Wrong Accept	#1310
LIBRESSL 4.3.1	Client	Wrong Alert	#1279
LIBRESSL 4.3.1	Server	Missing Alert	#1279
LIBRESSL 4.3.1	Server	Missing Alert	#1279
LIBRESSL 4.3.1	Server	Missing Alert	#1279
LIBRESSL 4.3.1	Client	Missing Alert	#1279
LIBRESSL 4.3.1	Client	Missing Alert	#1279
LIBRESSL 4.3.1	Client	Missing Alert	#1279
LIBRESSL 4.3.1	Client	Missing Alert	#1279
LIBRESSL 4.3.1	Clt+Srv	Missing Alert	#1279
LIBRESSL 4.3.1	Server	Missing Alert	#1279
LIBRESSL 4.3.1	Clt+Srv	Missing Alert	#1279
LIBRESSL 4.3.1	Server	Missing Alert	#1279
LIBRESSL 4.3.1	Clt+Srv	Missing Alert	#1279
LIBRESSL 4.3.1	Clt+Srv	Missing Alert	#1279
WOLFSSH 1.5.0	Server	Unexpected Msg	#1246
WOLFSSH 1.5.0	Server	Wrong Accept	#1287
WOLFSSH 1.5.0	Client	Wrong Reject	#1286
WOLFSSH 1.5.0	Client	Wrong Accept	#1285
LIBSSH 0.11.4	Server	Wrong Reject	#376

TABLE 2: Summary of the 54 bugs we discovered that were acknowledged by the vendors. Types name deviations from what the RFCs mandate, *e.g. Wrong Accept*: accepting a message that the RFCs require rejecting. *Security-Related* marks bugs with security implications.

5. DDYF and DYF can generate the same traces, but DYF silently drops traces triggering such CVEs as they do not violate DY security policies.

6. There is a priori no need to launch all pairwise combinations rather than comparing each PUT to a common one, as we did.

permissive default that OPENSSL's 2021 change later removed: in LIBRESSL the flag is enabled by default (an application must explicitly clear it to obtain the defense) and is, moreover, the entire content of `SSL_OP_ALL`, the benign-looking bug-compatibility macro most applications set—so the one option that could look like a hardening step instead re-arms the flag. This makes it a genuine "configuration trap": the defense is off unless the developer names and clears the exact flag, and enabling `SSL_OP_ALL` silently keeps it off. BORINGSSL, by contrast, removed the option altogether: the lenient behavior is hardwired, with no way for an application to request the defense. WOLFSSL, an independent implementation, implements the check but compiles it only under an opt-in hardening flag (`WOLFSSL_HARDEN_TLS`), so that default builds neither advertise nor require the extension. In each lenient case the weakened handshake still completes successfully, so a non-differential Objective Oracle has nothing to flag; only the divergence across implementations surfaces the gap (CWE-1188 for the three lenient stacks). Following our reports, LIBRESSL and WOLFSSL removed the insecure default and BORINGSSL restored strict enforcement as an opt-in.

The second gap, and fourth bug, is a missing second line of defense in LIBRESSL's Elliptic Curve code when checking group elements: on an adversarial degenerate input (the point at infinity), it performs the ECDH scalar multiplication with the private key before rejecting the point, rather than rejecting it up front. We verified this is not exploitable (the result is independent of the secret and the multiplication is constant-time), but it is a defense-in-depth gap. The developers acknowledged it and added an explicit point-at-infinity check before the multiplication; the same response also enforces uncompressed point encoding (rejecting the deprecated compressed and hybrid formats, RFC 8422) and now emits a fatal `Alert` on rejection in place of the previous silent close.

Other RFC Violations. Among the other RFC violations, 8 are *Wrong Accept* (7) or *Wrong Reject* (1) bugs, caused by missing or incorrect checks on incoming messages: the faulty PUT accepts a malformed and adversarial message it should reject as per the RFC, or rejects a valid message (oversized record, bad extension in a message, out of order messages, incorrect values, *etc.*). *E.g.* 4 such bugs concern `HelloRetryRequest` in WOLFSSL: a TLS 1.3 server accepts clients changing their chosen cipher when sending a second `ClientHello` and using a `KeyShare` group not requested in its `HelloRetryRequest` (though it is not possible to force the use of a key share group that the server does not support). A client also accepts a change of cipher between `HelloRetryRequest` and the following `ServerHello`, leading to an error when deriving the handshake keys. Also, when a client receives a `HelloRetryRequest` that contains only the `SupportedVersion` extension it sends a second identical `ClientHello`, which the RFC explicitly forbids. Moreover, LIBRESSL accepts a TLS 1.3-only cipher suite in a TLS 1.2 `ServerHello`. While these bugs could potentially lead to security issues, we failed to exploit any of them in practice.

The remaining ones are 20 *Missing Alert* bugs (silent aborts), where the PUT terminates the connection without sending any `Alert` while the RFC specifies that it MUST send one, 14 *Wrong Alert* bugs, where the PUT sends an `Alert` of the wrong type, and 1 *Duplicate Alert* bug, where it sends the same `Alert` twice. These bugs are not necessarily security-critical but can lead to interoperability issues and make debugging more difficult. They also show that all the PUTs we tested still fail to strictly enforce the state-machine specified in the TLS RFCs and could hint at potential attack vectors.

SSH RFC Violations. Although our experiments with SSH were not as extensive as for TLS, DDYF surfaced 5 RFC violations in two mature SSH stacks, see Table 2. For example, WOLFSSH wrongly sends port information in a `REQUEST_SUCCESS` message (*Unexpected Msg*) and LIBSSH rejects an identification banner larger than 129 bytes while the RFC allows up to 255 bytes (*Wrong Reject*). That said, one would need a longer campaign and a complete triaging to complete this SSH security analysis.

Takeaways. PUFFIN did not find critical zero-day vulnerabilities during those campaigns—although it found four security-related bugs, stemming from two defense-in-depth gaps in various stacks, all since addressed upstream—but did find known vulnerabilities for earlier versions of WOLFSSL (Section 5.3.1). Several factors may explain this. First, the PUTs we fuzzed are already extensively tested and audited. Second, the operators of the Mapper for building TLS messages in PUFFIN (the same as in [2]) are not exhaustive and cannot generate all TLS messages. It is thus possible that, although some new vulnerabilities can be detected by PUFFIN, they are only triggered by traces requiring operators that are not yet implemented; we plan to implement more operators in future work.

Yet, PUFFIN was able to find dozens of RFC violations—all acknowledged by the vendors and the majority already fixed upstream—requiring quite complex series of actions and adversarial messages that other fuzzers fall short of generating (see Section 6). By nature, such compliance and functional correctness bugs are difficult to find with non-differential techniques as this would require writing a specification of the expected behaviors. In contrast, our differential extension of PUFFIN is by design well-suited to find such bugs.

5.4. Evaluating the Effectiveness of DDYF

5.4.1. Fuzzer Accuracy (RQ2). We evaluate PUFFIN's accuracy by sorting the found objectives described in Section 5.3.2 into two categories: 1) RFC violations or security-related bugs; 2) specification-allowed differences (real differences due to different but compliant choices). The latter notably includes traces containing two or more errors: depending on the parse order PUTs may send different `Alert` errors. It also covers traces that exhibit different design choices, *e.g.* when receiving a `ClientHello`, an OPENSSL server selects the first supported cipher while WOLFSSL

selects its preferred one. We ran differential fuzzing campaigns for all supported PUTs (comparing each of them to OPENSSL) and measured the proportion of buckets in category 1): 3% (BORINGSSL), 12% (WOLFSSL), and 26% (LIBRESSL). The quite low ratio for BORINGSSL may be the consequence of several diverging choices unique to this stack and of its lack of an API to configure it and ease PUT alignment (see Section 4.2).

5.4.2. Performance and Ablation Study (RQ3,RQ4). In an experiment detailed in Appendix C we show that the Differential Oracle introduces no measurable slowdown, and that the only (expected) slowdown stems from executing two PUTs. We also conducted an ablation study to evaluate the effectiveness of each individual component of the Differential Oracle (status difference, knowledge difference, decryption, claim differences). We successively disabled each component and ran our triaging script on the same set of objectives to determine which proportion of differences is still found.

The results show that disabling claims, status, or knowledge results in losing some traces, which indicates that they all contribute detection capabilities to DDYF. Surprisingly, the decryption component did not find real differences on the TLS campaigns we ran; the only reported differences were false positives due to WOLFSSL including an optional extension and were discarded by our post-execution filter (Section 4.2). The decryption component might prove more useful in TLS when activating more TLS extensions (used in ciphertexts), and in other protocols. This component is not yet empirically proven to find bugs in TLS, even though lacking this ability would yield an obvious, large blind spot. However, the campaigns we ran over SSH did yield traces flagged exclusively by the decryption component, as well as evidence of RFC violations that we would have otherwise missed. More generally and fundamentally, it is the first time a protocol fuzzer can meaningfully and structurally compare data under encryption (see Section 6 for a comparison with [49]). Finally, we also observed that most differences are caught by several components, although each of the claims, status, and Knowledge comparisons is sometimes the only one to catch a difference (as is the decryption component in SSH campaigns). Note, however, that triaging greatly benefits from the precise difference descriptions that are only available when all components are enabled. Details are available in Appendix B.

TABLE 3: Stability of the OPENSSL 3.4.0 vs WOLFSSL 5.8.0 triaging over 10 campaigns (145 buckets in total)

	Mean	Min	Max	Std dev
Total traces	3561k	2678k	4576k	631k
Untriaged traces	2.0k (0.06%)	0.4k	2.9k	0.4k
Filled buckets	142	140	143	1.05

Note: Each of the 10 campaigns ran for 24h on 8 AMD EPYC 9275F CPU cores and was triaged into buckets by the LLM-produced triaging script.

5.4.3. Reliability of LLM Triaging.

Bucket coverage. To assess whether a triaging script produced with LLM assistance generalizes and does not leave blind spots, we ran 10 further differential campaigns between OPENSSL and WOLFSSL and applied the same script. Table 3 shows that on average only 0.06% of the traces are left untriaged, and that, in each campaign, between 97% and 99% of the 145 buckets are filled. Only one bucket was found in none of the 10 campaigns, due to the low probability of producing traces that match it. We thus conclude that our triaging yields robust and comprehensive coverage of the differences.

False positives and false negatives. A false positive is a bucket assessed as a bug that is not one; a false negative is a genuine bug that ends up in a bucket assessed as benign. Quantifying the false positives of the LLM alone is hard due to the interactive nature of the triaging: after each LLM triaging pass, each bucket is manually reviewed to spot suspicious ones (*e.g.* vague conditions or duplicates), which leads to another triaging round by the LLM. Once the triaging criteria appear satisfactory, all buckets are assessed for security implications or specification violations. This step is manual and requires reproducing the bug on a standalone implementation, checking the specification for an explicit violation, and reporting it to the maintainers. These manual verifications act as guardrails against reporting false positives: all 54 bugs of Table 2 were acknowledged by the vendors.

As for false negatives, we had first triaged the TLS 1.3 OPENSSL-WOLFSSL differences manually and found 11 bugs. The LLM-assisted triaging, run from scratch and without any knowledge of those bugs, rediscovered all 11 and flagged them as RFC violations, *i.e.* 100% recall with respect to this human-only baseline. Moreover, all bucket assessments, including the benign ones, were manually audited (Section 4.3).

We emphasize that LLMs are used to speed up the triaging process rather than to replace human triaging: LLM-generated results must be thoroughly reviewed.

6. Related work

Differential testing. Existing differential testing techniques, while efficient at finding some classes of bugs, do not cover the same scope as us. This is either due to the detection capabilities of their oracles, or the limited expressiveness of their input generation, or both. Many differential testing tools only check for one type of difference, such as the return code of the PUT [17], or the message type, *e.g.* a `ServerHello` or an `Alert`, but not its content [45, 46]. This limits their scope to message flow violations. (The same limitation applies to work relying on inferring Finite State Machines (FSM) [33, 34] as the content and structure of messages is not compared.) Moreover, as opposed to DDYF, these prior works either lack the ability to compare multi-message flows [17, 45, 46], or are limited to a narrow part of the protocols, only testing the first round trip or only the server side [45, 46]. Hence, client code and Machine-in-the-Middle (MIM) scenarios are not tested.

Zhao *et al.* [49] have a more fine-grained oracle, which compares a string representation of PUT responses to some extent. However, it requires replacing random field values with constants. To do so, the authors partially mitigate this problem with custom, manually written parsing functions for some protocol message types and custom representations for each field. As a result, all fields are ignored by default except when custom functions were written. In stark contrast, DDYF benefits from the DY model and re-interpretation of bitstrings as structured terms for free and thus defaults to comparing every type and ignoring only a few user-specified fields. As an illustration, Zhao *et al.* compare lists by comparing their size and not their content. The other types of information DDYF compares, *e.g.* Claims, are ignored as well. Their work [49] is the only one to provide (partial) support for decryption. However, support is limited to TLS 1.2 (where only the `Client/ServerFinished` are encrypted) for attacking client or server but not when the fuzzer acts as a MIM (due to the absence of Claims). Finally, we were able to confirm that at least 7 of the WOLFSSL RFC violations we found affected versions tested by [49] but were not detected, possibly due to scope limitations.

TLS testing. Several tools are specialized to TLS implementations. TLSAttacker [43, 8] allows creating new message flows and modifying messages to trigger erroneous behaviors. Such behaviors are caught by tools like ASAN or by specification-violation test suites. Similarly, [36] relies on test templates, verifying some RFC requirements, that are mutated with combinatorial testing. As opposed to our approach, those tools are not fuzzers and thus do not generate random test cases that can trigger new unexpected behaviors.

FlexTLS [10] allows writing custom TLS scenarios and uses MiTLS as a backend to encode and send messages. This tool has been used to reproduce some TLS attack traces and serves as the base of the SmackTLS fuzzer [10], which can check for some TLS specification violations. However, those are limited to TLS error handling, *i.e.* checking that incorrect traces finish with an Alert message.

Those approaches are able to find bugs in TLS implementations, but are by design tied to TLS: applying their methodology to other protocols would require a complete rewrite. Their capabilities to detect protocol violations also heavily depend on manually crafted properties, either checking specific RFC requirements [43, 8, 36] or verifying whether old CVEs affect new implementations [43, 8, 10], which makes it difficult to be exhaustive.

7. Conclusion and future work

In this work we introduced DDYF, an effective method for fuzzing cryptographic protocols that avoids the need to write dedicated protocol-specific bug oracles. We implemented this approach and evaluated its effectiveness on the widely used OPENSSL, WOLFSSL, LIBRESSL, and BORINGSSL TLS implementations, as well as on the LIBSSH and WOLFSSH SSH implementations. DDYF found all 5 known CVEs from our curated benchmark and uncovered 54 new RFC violations, four of which are security-related.

Fuzzing a new protocol with DDYF requires providing a blacklist of randomized fields; this could be further automated. Our differential oracle could also be strengthened to check whether equality relations between ignored fields are the same on both sides. This way we could additionally detect when some agreement on randomized data (*e.g.* session key) is only violated on one side. We also plan to apply DDYF to new protocols and implementations such as WireGuard, WPA2 or Bluetooth.

Finally, we would like to explore DDYF-based fingerprinting. Indeed, DDYF can expose fine-grained behavioral discrepancies, which would enable precise fingerprinting of protocol implementations.

Acknowledgment. The authors would like to thank Nataël Baffou for help on engineering efforts and fruitful discussions. Moreover, this work benefited from funding managed by the French National Research Agency under the France 2030 program (ANR-22-PECY-0006), and ANR JCJC project ProtoFuzz (ANR-22-CE48-0017).

References

[1] *aflnwe*. https://github.com/thuanpv/aflnwe.

[2] M. Ammann, L. Hirschi, and S. Kremer. "DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing". In: *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024.

[3] L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. A. Basin, and P. Müller. "Sound Verification of Security Protocols: From Design to Interoperable Implementations". In: *SP*. IEEE, 2023.

[4] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. K. äsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt. "DROWN: Breaking TLS Using SSLv2". In: *USENIX Security Symposium*. USENIX Association, 2016.

[5] 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.

[6] D. A. Basin, C. Cremers, J. Dreier, and R. Sasse. *Modeling and Analyzing Security Protocols with Tamarin - A Comprehensive Guide*. Springer, 2026.

[7] D. A. Basin, R. Sasse, and J. Toro-Pozo. "The EMV Standard: Break, Fix, Verify". In: *SP*. IEEE, 2021.

[8] F. Bäumer, M. Brinkmann, N. Erinola, S. Hebrok, N. Heitmann, F. Lange, M. Maehren, and R. Merget. "TLS-Attacker: A Dynamic Framework for Analyzing TLS Implementations". In: 2024.

[9] 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.

[10] B. Beurdouche, A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and K. Bhargavan. "FLEXTLS: A Tool for Testing TLS Implementations". In: *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*. USENIX Association, 2015.

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

[12] K. Bhargavan, V. Cheval, and C. A. Wood. "A Symbolic Analysis of Privacy for TLS 1.3 with Encrypted Client Hello". In: *CCS*. ACM, 2022.

[13] K. Bhargavan, C. Jacomme, F. Kiefer, and R. Schmidt. "Formal verification of the PQXDH Post-Quantum key agreement protocol for end-to-end secure messaging". In: *USENIX Security Symposium*. USENIX Association, 2024.

[14] K. Bhargavan, A. D. Lavaud, C. é. 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.

[15] B. Blanchet, V. Cheval, and V. Cortier. "ProVerif with Lemmas, Induction, Fast Subsumption, and Much More". In: *SP*. IEEE, 2022.

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

[17] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations". In: *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014.

[18] K. Claessen and J. Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 2000.

[19] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *CCS*. ACM, 2017.

[20] 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: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2016.

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

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

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

[24] 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.

[25] 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.

[26] D. Dolev and A. C. Yao. "On the security of public key protocols". In: *IEEE Trans. Inf. Theory* 29.2 (1983).

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

[28] 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.

[29] 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.

[30] 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.

[31] T. Gouville, L. Hirschi, and S. Kremer. "DDYF: Differential Dolev-Yao Fuzzing of Cryptographic Protocols". In: *SP*. IEEE, 2027.

[32] T. Gouville, L. Hirschi, and S. Kremer. *Paper artifact: DDYF implementation, reproducible evaluation, and list of bugs*. Software Heritage: swh:1:dir:3d2794a0befb86fae51fef18c03ed4ee681e69b3.

[33] 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: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. event-place: Virtual Event, Republic of Korea. Association for Computing Machinery, 2021.

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

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

[36] 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. USENIX Association, 2022.

[37] V. J. Manès, 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* 47.11 (2019).

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

[39] K. G. Paterson and T. van der Merwe. "Reactive and Proactive Standardisation of TLS". In: *SSR*. Vol. 10074. Springer, 2016.

[40] 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.

[41] 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: *Computer Security – ESORICS 2022*. Vol. 13556. Series Title: Lecture Notes in Computer Science. Springer Nature Switzerland, 2022.

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

[43] J. Somorovsky. "Systematic Fuzzing and Testing of TLS Libraries". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[44] J. Sun, T. Su, J. Jiang, J. Wang, G. Pu, and Z. Su. "Property-based fuzzing for finding data manipulation errors in android apps". In: *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*. 2023.

[45] A. Walz and A. Sikora. "Exploiting dissent: towards fuzzing-based differential black-box testing of TLS implementations". In: *IEEE Transactions on Dependable and Secure Computing* 17.2 (2017). Publisher: IEEE.

[46] A. Walz and A. Sikora. "Maximizing and Leveraging Behavioral Discrepancies in TLS Implementations using Response-Guided Differential Fuzzing". In: *2018 International Carnahan Conference on Security Technology (ICCST)*. 2018.

[47] J. Wilson and M. Asplund. "Analysing TLS Implementations Using Full-Message Symbolic Execution". In: *Secure IT Systems - 29th Nordic Conference, NordSec 2024 Karlstad, Sweden, November 6-7, 2024 Proceedings*. Vol. 15396. Springer, 2024.

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

[49] Z. Zhao, X. Song, Q. Zhong, Y. Zeng, C. Hu, and S. Guo. "TLS-DeepDiffer: Message Tuples-Based Deep Differential Fuzzing for TLS Protocol Implementations". In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024.

Appendix A.
Integration Effort of TLS PUTs for Differential Fuzzing in PUFFIN

DDYF and PUFFIN are protocol-agnostic and can be specialized to a new protocol by implementing the protocol-specific PUFFIN interface, see Section 5.1.2.

Adding DDYF support for TLS represents ca 1.8 kLoC including adjustments to align WOLFSSL, OPENSSL, BORINGSSL and LIBRESSL: roughly 200 LoC for each PUT (aligning their behaviors) and the remaining 1 kLoC for the protocol-layer. In particular, we added new interfaces to set cipher suites and curves for TLS 1.3 and TLS 1.2 and selected two TLS 1.3 ciphers, two TLS 1.2 ciphers and two elliptic curves that are supported by all implementations. PUFFIN already provides convenient code annotations to specify how protocol messages can be partially parsed (for defining $\lfloor \cdot \rfloor$). We extended this PUFFIN API to additionally provide the blacklist for ignoring some messages and claims during comparison, as well as a way to provide custom comparison functions. We also wrote two decryption recipes (one for decrypting the first server encrypted flight and one to decrypt the client's finished message). Also, we updated PUFFIN TLS Claims to access the handshake secrets for the decryption recipes and added new operators for the new encryption, decryption, and flight abstractions. Some of the previous TLS seeds needed to be adapted to use our new message flight abstraction and to align the behaviors of the PUTs.

Appendix B.
Ablation Study

We ran the ablation study on the objectives of our TLS campaigns (one per pair of PUTs). Table 4 reports the number of traces that are still flagged when disabling a component, or enabling only a specific component. Note that while the DDYF campaigns we ran on TLS produced no traces for which decryption recipes were absolutely necessary for being flagged (No decryption: -0), the campaigns we ran over SSH did yield traces flagged exclusively by the decryption component, as well as evidence of RFC violations that we would have otherwise missed.

Appendix C.
Performance Study

DDYF requires executing each trace twice and comparing the results of both executions. Since executing a trace on a single PUT takes 84% of DYF time [2, §6.1.2], we expect almost a 50% slowdown for DDYF. We measure the average number of executions per second of PUFFIN in differential mode over 10 runs of 1 hour on 1 core

TABLE 4: Ablation study results on TLS differential campaigns run against OPENSSL, counting found objectives

Components	BORINGSSL	LIBRESSL	WOLFSSL
All	1322k	2681k	3713k
No claims	1299k (-23k)	2669k (-12k)	3694k (-19k)
No decryption	1322k (-0)	2681k (-0)	3713k (-0)
No knowledges	853k (-469k)	2362k (-319k)	3252k (-460k)
No status	1148k (-174k)	2267k (-414k)	3121k (-592k)
Only claims	120k (-1202k)	142k (-2539k)	906k (-2807k)
Only knowledges	1124k (-198k)	2244k (-437k)	3084k (-629k)
Only status	822k (-500k)	2343k (-338k)	3205k (-508k)

between WOLFSSL 5.8.0 and OPENSSL 3.4.0 and compare it with PUFFIN running one of the PUTs[7]. The results are displayed in Table 5. These experiments show no measurable slowdown of the Differential Oracle, compared to executing two PUTs.

TABLE 5: Performance of DDYF vs standard DYF in PUFFIN (1 core / 1h)

Configuration	Mean execs per second per core	Std. dev
DYF OPENSSL	552 exec/s	51 exec/s
DYF WOLFSSL	687 exec/s	92 exec/s
DDYF OPENSSL vs OPENSSL	330 exec/s	84 exec/s
DDYF WOLFSSL vs WOLFSSL	271 exec/s	11 exec/s
DDYF OPENSSL vs WOLFSSL	321 exec/s	21 exec/s

Appendix D. Deduplication Performance

Figure 4 shows the impact of objective deduplication on a 24h fuzzing campaign between OPENSSL and WOLFSSL. Out of the ca 68M traces flagged as objectives during the campaign, ca 64M were discarded as duplicates, leaving ca 3.7M objectives, for a final corpus of ca 10k traces. We replicated this experiment 10 times and measured an average duplicate proportion of 94.8% (min: 93.8%, max: 95.6%, std. dev.: 0.006) and 4.2M objectives (std. dev.: 542k) at the end of the campaigns.

We observe that the proportion of duplicates increases as the corpus saturates. This is expected as a trace is considered a duplicate when it has the same set of differences and PUT statuses and originates from the same corpus trace. At the beginning of the fuzzing campaign, objectives originating from new corpus traces are not considered duplicates, but once the corpus saturates, mutations on the same corpus trace tend to create multiple objectives with the same set of differences, *i.e.* duplicates.

Note that fuzzing inherently produces many traces that are not interesting and are never promoted to the corpus. In differential fuzzing, however, objectives are much more prevalent since any difference is flagged as an objective: the same difference is thus expected to be hit many times,

7. Figures slightly differ from [2] due to experiment differences.

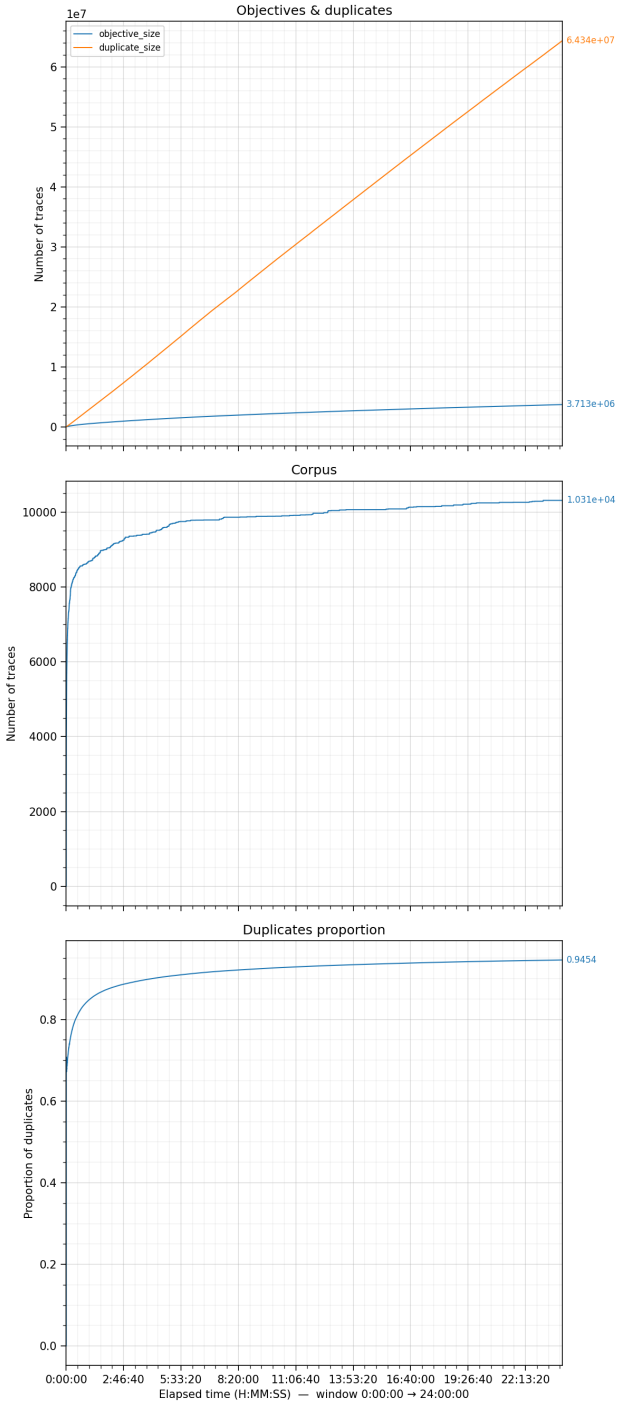

Figure 4: Deduplication of objectives

which further motivates the need for effective deduplication. Improving the efficiency and diversity of trace generation, which we inherit from DYF, is however out of the scope of this paper.

We also verified that no objective diversity was lost through this mechanism: we triaged some campaigns without deduplication to create a set of buckets, then ran a

campaign on the same PUTs with deduplication to confirm that we were still able to find traces for every bucket.

Appendix E.
Details of the Acknowledged Bugs

Table 6 describes the bugs of Table 2, which we reported to the PUT maintainers and which they acknowledged, and Table 7 summarizes them per implementation and type.

TABLE 6: Description of the 54 acknowledged bugs listed in Table 2. The bug types are defined in Section 5.3.2.

PUT Version	Agent	Type	Description
wolfSSL 5.9.1	Client	Wrong Alert	wolfSSL TLS 1.3 client receiving a HelloRetryRequest (HRR) with mandatory `SupportedVersions` extension missing sends `ProtocolVersion` alert instead of `MissingExtension`. Issue
wolfSSL 5.9.1	Client	Wrong Alert	wolfSSL TLS 1.3 client receiving a `ServerHello` with `legacy_version = 0x0303` (TLS 1.2) and no `SupportedVersions` extension sends a `DecodeError` alert instead of `ProtocolVersion`. Issue
wolfSSL 5.9.1	Client	Wrong Alert	wolfSSL TLS 1.3 client receiving a `ServerHello` whose random field equals the `HelloRetryRequest` magic value and whose extensions include a recognized extension that is not permitted in a tlsTermHelloRetryRequest sends an `UnsupportedExtension` alert instead of `IllegalParameter`. Issue
wolfSSL 5.9.1	Client	Duplicate Alert	wolfSSL TLS 1.3 client receiving a `HelloRetryRequest` with mandatory `SupportedVersions` extension missing sends two identical alert messages. Issue
wolfSSL 5.9.1	Client	Missing Alert	wolfSSL TLS 1.3 client silently rejects a TLS record whose length exceeds the 2^{14}-byte limit mandated by RFC 8446, without sending a `RecordOverflow` alert. Issue
wolfSSL 5.8.2	Client	Wrong Accept	When receiving a `HelloRetryRequest` (`ServerHello` with magic random value) which should trigger no change in the `ClientHello` (the `HRR` only contains a `SupportedVersion` extension), wolfSSL TLS 1.3 client sends a `ClientHello` that is identical to its first `ClientHello`. Issue
wolfSSL 5.8.2	Server	Wrong Accept	wolfSSL can respond to a `ClientHello` without a `SupportedGroup` extension if the `ClientHello` contains a `KeyShare`. This behavior happens only when setting the list of supported groups on the server using `wolfSSL_set_groups` or `wolfSSL_CTX_set1_groups_list` or using `--force-curve` in the example CLI. Issue
wolfSSL 5.8.2	Client	Wrong Accept	A wolfSSL TLS 1.3 client receiving a `HelloRetryRequest` followed by a `ServerHello` with another cipher (*e.g.* using `TLS_AES_256_GCM_SHA384` in `HRR` and `TLS_AES_128_GCM_SHA256` in `ServerHello`), will continue the handshake while it should reject the `ServerHello`. Issue
wolfSSL 5.8.2	Server	Wrong Accept	A wolfSSL TLS 1.3 server sending a `HelloRetryRequest` requesting a specific `KeyShare` can accept a second `ClientHello` with a `KeyShare` not present in the `HelloRetryRequest`. Issue
wolfSSL 5.8.2	Client	Wrong Alert	A wolfSSL TLS 1.3 client receiving a `ServerHello` with a `PresharedKey` extension when the client has not requested the use of a preshared key, returns an `IllegalParameter Alert` instead of an `UnsupportedExtension Alert`. Issue
wolfSSL 5.8.4	Server	Wrong Accept	A wolfSSL TLS 1.3 server receiving a `ClientHello` without a `KeyShare` extension will select one of the offered ciphers of the client and send a `HelloRetryRequest`, it will then accept a second `ClientHello` with a list of ciphers not selected in the `HelloRetryRequest` and use this new cipher in subsequent messages. Issue
wolfSSL 5.8.4	Server	Wrong Alert	A wolfSSL TLS 1.3 server receiving a `ClientHello` without a `SupportedGroup` extension if the `ClientHello` contains a `KeyShare`, returns an `HandshakeFailure Alert` instead of a `MissingExtension Alert`. Issue
wolfSSL 5.8.4	Client	Missing Alert	A wolfSSL TLS 1.3 client receiving a `ServerHello` containing a duplicated extension (*e.g.* two `SupportedVersion`s) aborts the connection but does not send an `Alert` instead of returning an `IllegalParameter Alert`. Issue
wolfSSL 5.8.4	Client	Missing Alert	A wolfSSL TLS 1.3 client receiving encrypted out-of-order messages will detect the error and log "`Out of order message, fatal`" without sending an `Alert`. Issue

TABLE 6: Description of the 54 acknowledged bugs (continued)

PUT Version	Agent	Type	Description
WOLFSSL 5.8.4	Client	Wrong Alert	A WOLFSSL TLS 1.3 client receiving a `ServerHello` with a cipher suite that was not offered in the `ClientHello` returns a `HandshakeFailure Alert` instead of an `IllegalParameter Alert`. Issue
WOLFSSL 5.8.4	Server	Wrong Alert	A WOLFSSL TLS 1.3 server receiving a `ClientHello` with a malformed `KeyShare` extension (*e.g.* with no key inside) returns an `IllegalParameter Alert` instead of a `DecodeError Alert`. Issue
WOLFSSL 5.8.4	Client	Security-Related	A WOLFSSL TLS 1.2 client accepts a `ServerHello` without the renegotiation extension, thus not providing cryptographic binding between sessions which allows for a CVE-2009-3555-style splice attack. Reported by email. Upstream fix
LIBRESSL 4.3.1	Server	Wrong Alert	LIBRESSL server sends `IllegalParameter` instead of mandatory `MissingExtension` for absent mandatory TLS 1.3 `SupportedGroup` extension in `ClientHello`. Issue
LIBRESSL 4.3.1	Server	Missing Alert	LIBRESSL TLS server silently closes on out-of-order handshake message instead of sending `UnexpectedMessage` alert. Issue
LIBRESSL 4.3.1	Client	Missing Alert	LIBRESSL TLS client silently closes on out-of-order handshake message instead of sending `UnexpectedMessage` alert. Issue
LIBRESSL 4.3.1	Client	Missing Alert	LIBRESSL silently aborts on `HelloRetryRequest` with malformed `KeyShare` payload length instead of sending `DecodeError`. Issue
LIBRESSL 4.3.1	Client	Missing Alert	LIBRESSL silently aborts on invalid ECDH public key instead of sending `IllegalParameter` alert. Issue
LIBRESSL 4.3.1	Clt+Srv	Missing Alert	LIBRESSL silently aborts on TLS version negotiation mismatch instead of sending `ProtocolVersion` alert. Issue
LIBRESSL 4.3.1	Client	Missing Alert	LIBRESSL silently aborts when `HelloRetryRequest` selects a group the client never offered. Issue
LIBRESSL 4.3.1	Client	Wrong Accept	LIBRESSL client accepts TLS 1.3-only cipher in a TLS 1.2 `ServerHello`. Issue
LIBRESSL 4.3.1	Client	Wrong Alert	LIBRESSL client receiving two `HelloRetryRequest` in a row returns the wrong alert. Issue
LIBRESSL 4.3.1	Server	Missing Alert	LIBRESSL TLS 1.2 server receiving a record whose type is invalid for its current state closes the connection without sending the RFC-mandated `UnexpectedMessage` alert. Issue
LIBRESSL 4.3.1	Server	Missing Alert	LIBRESSL TLS 1.3 server silently aborts on malformed `ClientHello KeyShare` length. Issue
LIBRESSL 4.3.1	Client	Security-Related	A LIBRESSL TLS 1.3 client receiving the point at infinity in the server `KeyShare` performs the ECDH scalar multiplication with its private key before rejecting the point, and then silently aborts. Reported by email. Upstream fix #1 Upstream fix #2
LIBRESSL 4.3.1	Server	Missing Alert	LIBRESSL server silently aborts on missing `SignatureAlgorithms`. Issue
LIBRESSL 4.3.1	Client	Missing Alert	LIBRESSL client silently aborts on oversized `TLSCiphertext`. Issue
LIBRESSL 4.3.1	Client	Missing Alert	LIBRESSL client silently rejects malformed `CertificateRequest` (CA names / `CertificateRequest` length mismatch). Issue
LIBRESSL 4.3.1	Clt+Srv	Missing Alert	LIBRESSL silently aborts on oversized handshake messages. Issue
LIBRESSL 4.3.1	Server	Missing Alert	LIBRESSL server silently aborts when it receives an unchanged `ClientHello` after a `HelloRetryRequest`. Issue
LIBRESSL 4.3.1	Clt+Srv	Missing Alert	LIBRESSL server silently aborts when receiving a message with incorrect protocol version. Issue
LIBRESSL 4.3.1	Server	Missing Alert	LIBRESSL server silently closes when receiving a `ClientHello` where no key share matches the server's supported groups. Issue
LIBRESSL 4.3.1	Clt+Srv	Missing Alert	LIBRESSL silently aborts when receiving malformed or unsolicited extension. Issue

TABLE 6: Description of the 54 acknowledged bugs (continued)

PUT Version	Agent	Type	Description
LIBRESSL 4.3.1	Client	Missing Alert	LIBRESSL silently aborts when a `ServerHello` with an invalid cipher suite. Issue
LIBRESSL 4.3.1	Client	Missing Alert	LIBRESSL TLS 1.3 client silently aborts upon receiving a `HelloRetryRequest` requesting a `KeyShare` that the client has already provided. Issue
LIBRESSL 4.3.1	Client	Missing Alert	LIBRESSL TLS 1.3 client silently aborts upon receiving a second malformed `HelloRetryRequest` (bad `LegacyVersion`, cipher change and no extensions). Issue
LIBRESSL 4.3.1	Client	Security-Related	A LIBRESSL TLS 1.2 client with default configuration accepts a `ServerHello` without the renegotiation extension, thus not providing cryptographic binding between sessions which allows for CVE-2009-3555-style splice attack. Reported by email. Upstream fix
OPENSSL 3.6.0	Client	Wrong Alert	An OPENSSL TLS 1.3 client receiving a `ServerHello` with a `StatusRequest` extension returns an `UnsupportedExtension Alert` instead of an `IllegalParameter Alert`. Issue
OPENSSL 3.6.2	Client	Wrong Alert	An OPENSSL TLS 1.3 client receiving a `ServerHello` without a `KeyShare` extension returns an `IllegalParameter Alert` instead of a `MissingExtension Alert`. Issue
OPENSSL 3.4.0	Client	Wrong Reject	An OPENSSL client rejects 0-RTT payloads of more than 103 bytes due to an overflow. Issue
OPENSSL 3.4.0	Client	Wrong Accept	An OPENSSL client accepts `HelloRequest` messages, which are explicitly forbidden by RFC 8446, without aborting the connection. Issue
BORINGSSL 0.20260327	Server	Wrong Alert	A BORINGSSL TLS 1.3 server receiving a `ClientHello` without a `SupportedGroup` extension returns a `HandshakeFailure Alert` instead of a `MissingExtension Alert`. Issue
BORINGSSL 0.20260327	Client	Security-Related	A BORINGSSL TLS 1.2 client accepts a `ServerHello` without the renegotiation extension, thus not providing cryptographic binding between sessions which allows for CVE-2009-3555-style splice attack. Reported by email. Upstream fix #1 Upstream fix #2
BORINGSSL 0.20260526	Server	Wrong Alert	A BORINGSSL TLS 1.3 server receiving a `ClientHello` without a `KeyShare` extension returns a `HandshakeFailure Alert` instead of a `MissingExtension Alert`. Issue
BORINGSSL 0.20260526	Client	Wrong Alert	A BORINGSSL TLS 1.3 client receiving a `ServerHello` with a non-authorized extension returns an `UnsupportedExtension Alert` instead of an `IllegalParameter Alert`. Issue
WOLFSSH 1.5.0	Server	Unexpected Msg	When a WOLFSSH server accepts an `SSH_MSG_GLOBAL_REQUEST` "tcpip-forward" whose requested bind port is non-zero and `want_reply` is true, it replies with `SSH_MSG_REQUEST_SUCCESS` followed by a uint32 bound-port field instead of only sending `SSH_MSG_REQUEST_SUCCESS` with no response-specific data. Issue
WOLFSSH 1.5.0	Server	Wrong Accept	A WOLFSSH server receiving an `SSH_MSG_USERAUTH_REQUEST` with valid credentials from a client that never sent `SSH_MSG_SERVICE_REQUEST` accepts it and replies with `SSH_MSG_SERVICE_ACCEPT` for a service that was never requested (RFC 4253 §10). Issue
WOLFSSH 1.5.0	Client	Wrong Reject	A WOLFSSH client cannot authenticate by public key against a server that does not send `SSH_MSG_EXT_INFO` with `server-sig-algs`, although RFC 8308 makes this message optional: it sends no public-key `SSH_MSG_USERAUTH_REQUEST` and aborts. Issue
WOLFSSH 1.5.0	Client	Wrong Accept	A WOLFSSH client that sent `SSH_MSG_SERVICE_REQUEST` accepts an unsolicited `SSH_MSG_USERAUTH_SUCCESS` in place of `SSH_MSG_SERVICE_ACCEPT`, and reports a successful connection to the application although no authentication took place (RFC 4252 §5.1). Issue
LIBSSH 0.11.4	Server	Wrong Reject	LIBSSH's server rejects client identification strings that are RFC 4253-compliant. Issue

Appendix F.
Ethical Considerations

The PUFFIN fuzzer has the potential to identify new bugs, vulnerabilities, and implementation discrepancies in cryptographic protocols. We designed and evaluated these tools in a way that limits opportunities for misuse during our own study. However, like any security analysis tool, they could potentially be misused by others; for instance, to find and exploit vulnerabilities or to perform stealthy reconnaissance before an attack. Nevertheless, we believe that the benefits to the community—providing developers with automated tools to proactively audit complex logical flaws, ensuring rigorous RFC compliance across the ecosystem, and enabling researchers to perform analyses and measurements of the global Internet infrastructure—significantly outweigh the risks of misuse.

To mitigate potential harms during our own research, we emphasize that *no live systems or real URLs were probed*. Furthermore, we have followed the principles of responsible disclosure for the bugs we discovered that we consider to be security-related or significant RFC violations. We always respected reasonable embargo periods and worked closely with the developers during this time. By disclosing these findings, we affirmatively protect users by allowing vendors to patch vulnerabilities before they can be exploited in the wild.

Appendix G.
List of Changes

- v1.0, May 18, 2026: Initial version.
- v2.0, October 14, 2026: Demonstrating generality of DDYF with new support for SSH (LIBSSH and WOLFSSH). LLM-assisted triaging. Removal of fingerprinting application to sharpen focus on DDYF. New deduplication mechanism and evaluation thereof. Complete DDYF analysis of BORINGSSL and LIBRESSL. Extended evaluation has additionally found 41 new RFC violations, including 4 security-related bugs. Major rewriting.

PUT	Total	Missing Alert	Wrong/Duplicate Alert	Security-Related	Wrong Accept	Wrong Reject	Unexpected Msg
LIBRESSL	24	19	2	2	1	0	0
WOLFSSL	17	3	8	1	5	0	0
OPENSSL	4	0	2	0	1	1	0
BORINGSSL	4	0	3	1	0	0	0
WOLFSSH	4	0	0	0	2	1	1
LIBSSH	1	0	0	0	0	1	0
Total	54	22	15	4	9	3	1

TABLE 7: Number of acknowledged bugs per implementation and bug type. The bug types are defined in Section 5.3.2.