Exploiting Dissent: Towards Fuzzing-based Differential Black-Box Testing of TLS Implementations

Andreas Walz, Axel Sikora

Abstract—The Transport Layer Security (TLS) protocol is one of the most widely used security protocols on the internet. Yet do implementations of TLS keep on suffering from bugs and security vulnerabilities. In large part is this due to the protocol's complexity which makes implementing and testing TLS notoriously difficult. In this paper, we present our work on using differential testing as effective means to detect issues in black-box implementations of the TLS handshake protocol. We introduce a novel fuzzing algorithm for generating large and diverse corpuses of *mostly-valid* TLS handshake messages. Stimulating TLS servers when expecting a ClientHello message, we find messages generated with our algorithm to induce more response discrepancies and to achieve a higher code coverage than those generated with American Fuzzy Lop, *TLS-Attacker*, or *NEZHA*. In particular, we apply our approach to *OpenSSL*, *BoringSSL*, *WolfSSL*, *mbedTLS*, and *MatrixSSL*, and find several real implementation bugs; among them a serious vulnerability in *MatrixSSL 3.8.4*. Besides do our findings point to imprecision in the TLS specification. We see our approach as present in this paper as the first step towards fully interactive differential testing of black-box TLS protocol implementations. Our software tools are publicly available as open source projects.

Index Terms—TLS, network security, cryptographic protocols, fuzzing, differential testing.

1 Introduction

In an increasing number of domains secure network communication is a key requirement, not only for functional safety but also for user acceptance. This is particularly true in the *Internet of Things* (IoT), where *Cyber Physical Systems* connect the physical world with the worldwide cyber space.

The *Internet Engineering Task Force* has developed the *Transport Layer Security* (TLS) protocol to provide *end-to-end* security over insecure networks [1], [2]. Since quite some time TLS is one of the Internet's cornerstones for secure communication. TLS supports encrypted and integrity-checked data transfer as well as mutual authentication of communication partners.

The TLS protocol is receiving continual scrutiny [3], [4], [5], [6], [7] and over time a long list of serious attacks on the protocol and some of its cryptographic constructions have been found [8], [9]. Still, in suitable and up-to-date configurations, TLS as a protocol can be considered sound and secure [10].

However, security vulnerabilities do not only arise from the protocol itself. Often, it is *the implementation* of the protocol which introduces serious security issues [11], [12]. This fact is emphatically demonstrated by another long list of security vulnerabilities, this time caused by flaws in popular implementations of TLS, e.g. the *Heartbleed bug* [13] and the *CCS injection bug* [14] in *OpenSSL* [15], the *goto fail bug* [16] in *GnuTLS* [17], and others.

• *The authors are with the Institute of Reliable Embedded Systems and Communication Electronics (ivESK), Offenburg University of Applied Sciences, Germany. Email: {andreas.walz, axel.sikora}@hs-offenburg.de.*

Second revision of manuscript TDSC-2017-01-0005

There are several reasons that make TLS implementations particularly susceptible to bugs:

- **Flexibility and agility:** The TLS protocol has a large parameter space with dynamic negotiation mechanisms. Several extension points allow adding new cryptographic algorithms and protocol features without touching the core specification.
- **Complexity:** The protocol's flexibility brings a high complexity to the protocol state machine, the handling of interactions between different protocol modes, and the message parsing. For example, TLS requires context-sensitive message parsers which are known to be a common source of bugs [18]. The parsing routines of TLS implementations can easily exceed several thousand lines of code.
- **Specification:** The specification of TLS along with all its extensions and options is distributed over nearly one hundred documents with thousands of pages. Beyond that does the specification not have formal character, lacking precision and leaving interpretation at the developer's discretion.
- **Interoperability and diversity:** Intolerant behavior in the diverse corpus of internet-deployed TLS implementations forces developers to individually trade strict conformance to the specification against large-scale interoperability.

Unfortunately does testing TLS implementations turn out to be highly nontrivial; at least if more than mere *mostly-positive* testing is aimed for [19]. Some of the reasons are:

- **Protocol messages:** The complexity and peculiarity of the TLS message format demand sophisticated test message generators. TLS features a multitude of different message, content, and data types. Ignorantly generated test messages are likely to fail early input validation.

- **State machine:** The TLS protocol state machine requires dynamic and nontrivial protocol interactions for deep and thorough probing.
- **Cryptography:** By its nature, TLS makes extensive use of cryptography. The use of cryptography requires elaborate and dynamic test interactions, just like a complex state machine does.
- **Lack of formal reference:** Given the lack of a formal TLS specification, no entity exists that can reliably tell correct from incorrect implementation behavior. After all, who can reliably decide whether the outcome of a test is positive or negative?

In particular the last-mentioned issue is addressed by various research activities: for example, *miTLS* is a reference implementation of TLS verified with respect to certain security properties [20], [21]. However, it is unlikely that verified implementations like *miTLS* fully replace prevalent implementations like *OpenSSL*, *BoringSSL*, or *WolfSSL* in the near future. In particular, this cannot be expected where severely resource-constrained devices, e.g. IoT nodes, are involved. Hence, the need for appropriate testing concepts and tools apparently still persists.

Motivated by this, we present our work on designing and implementing a novel test methodology that aims to help developers and users to identify issues in black-box TLS implementations. Our work has been inspired by *Frankencerts* [22], an approach applying fuzzing techniques and differential testing to the X.509 certificate validation routines that the authentication in TLS is based on. In contrast to *Frankencerts*, however, we put our focus on the implementation of the TLS handshake protocol itself.

Briefly explained, we use semi-randomly generated TLS protocol messages to stimulate multiple TLS implementations with equivalent input and use discrepancies in their responses to detect implementation bugs. Our approach does neither require code instrumentation nor execution monitoring, a property particularly interesting for developers of embedded platforms. At the heart of our approach is a novel fuzzing algorithm that generates TLS protocol messages which are highly effective in triggering discrepant protocol behavior among different TLS implementations.

Clearly, differential testing itself is not a new concept [23]. However, to the best of our knowledge, we are the first to apply fuzzing-based differential testing to the TLS handshake protocol itself. In previous work, differential testing has only been applied to the validation of certificates [22], [24], [25], [26].

We see several use cases for our approach:

- **Debugging and nonconformance detection:** Comparing a set of different TLS implementations among each other or comparing an implementation under test against a reference implementation is a way to detect implementation bugs. Moreover may it expose behavior that does not conform to the specification.
- **Regression testing:** Two versions of the same protocol implementation might be contrasted in order to gain confidence that, while changes and improvements have been implemented, no unintended change of behavior has been introduced.
- **Fingerprinting:** Comparing an unknown remote protocol implementation to a set of known reference implementations might allow to infer the origin of the unknown protocol implementation.
- **Identifying imprecise standardisation:** When comparing a set of protocol implementations, disagreement among the implementations may serve as a pointer to an imprecise protocol specification.

This paper presents the following contributions:

- **Generic Message Trees:** We design and implement the concept of *Generic Message Trees* (GMTs), a versatile and dynamic data structure for efficiently operating on highly-structured protocol messages.
- **Fuzzing-based TLS message generation:** Based on the GMT concept, we present a randomized algorithm for generating highly diverse and *mostly-valid* TLS handshake messages. Mostly-valid means that messages tend to obey all except very few syntactic and semantic rules.
- **Differential testing of ClientHello parsing:** We use our message generation algorithm to differentially test five popular TLS server implementations: *OpenSSL*, *BoringSSL*, *WolfSSL*, *mbedTLS*, and *MatrixSSL*. In that course, we show that our algorithm is more effective in provoking discrepant resonses than American Fuzzy Lop [27], *TLS-Attacker* [28], and *NEZHA* are. We study some of the bugs we exposed in TLS implementations using our test approach.
- **Software framework:** We make our software available for other researchers and developers [29].

Currently, our approach is limited to challenging the parsing routines for ClientHello messages embedded in TLS server implementations. Note that during this initial phase of the TLS handshake cryptography does not pose a challenge. However, an important goal of our ongoing work is to overcome these limitations and allow to test the whole handshake process, including the client perspective.

Our paper is organized as follows: Section 2 gives an overview of the TLS protocol, while Section 3 outlines related work. The concept of GMTs is presented in Section 4. Section 5 explains our test methodology, while its evaluation is provided in Section 6. A study of some TLS implementation bugs that we found is given in Section 7. Finally, Section 8 sketches future research directions and Section 9 summarizes the paper.

2 The TLS Protocol Family

The TLS protocol provides a flexible framework for cryptographically securing communication from end to end over networks assumed to be under an active attacker's control. Since its outset, TLS evolved from version 1.0 [30] to its

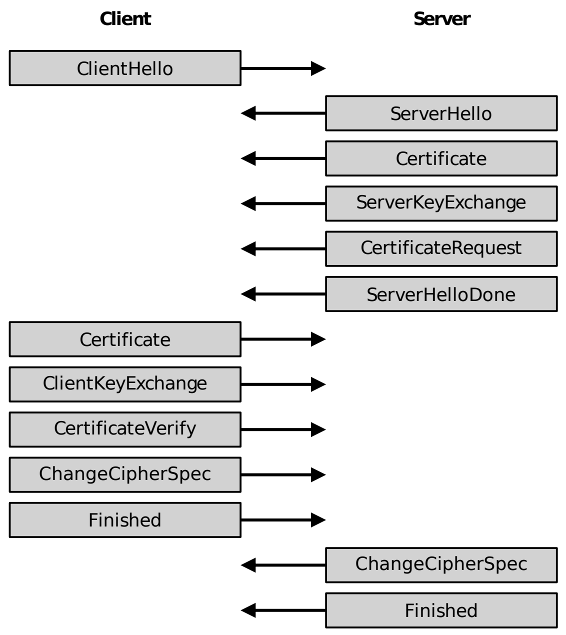


Fig. 1. Illustration of the sequence of TLS protocol messages typically exchanged between a TLS client and a TLS server during a handshake with mutual authentication. Depending on the authentication and key agreement scheme, some message may be optional or have a context-dependent interpretation. Note that with the upcoming TLS 1.3 the handshake is going to change significantly [31].

currently latest version 1.2 [1]. At present, version 1.3 of TLS is under preparation [31].

TLS is composed of five sub-protocols. The *Record Protocol* fragments data from higher TLS layers and handles data encryption and integrity protection. Its protocol data unit is called a *TLS record*. Stacked on top are the *Handshake*, *Change Cipher Specification*, *Alert*, and *Application Data* protocols. The first two are responsible for negotiating connection parameters and cryptographic keys. The Alert Protocol delivers error messages and the Application Data Protocol is used to securely transfer application data.

A TLS connection is established via a handshake where client and server exchange multiple messages (see Fig. 1). Each handshake message comprises a complex structure of binary-encoded integer, enumeration, and opaque fields as well as nested sub-structures. A client initiates a TLS handshake by sending a ClientHello message with a set of supported protocol options. If possible, the server responds with a ServerHello message and conveys its respective choice of options. Otherwise, the server typically responds with an Alert messages.

Authentication of the server (and optionally the client) as well as the establishment of a shared cryptographic secret is achieved by exchanging *Certificate* and *KeyExchange* messages. Finally, *Finished* messages complete the handshake and confirm its authenticity in retrospect by means of a signature over a transcript of the full handshake. Several extension points in the messages allow to retrofit new protocol features.

3 Related Work

Various generic black-box approaches use a model of the system under test – either learned from dynamic interactions or provided as additional input – to find vulnerabilities in network protocol implemenations [32], [33], [34], [35], [36]. However, the applicability of such approaches tends to scale poorly with the complexity of the protocol under consideration; a fact that seems prohibitive for TLS.

De Ruiter and Poll use state machine learning to infer high-level states and transitions in black-box TLS implemenation [37]. Based on visualizations of the learned state machines bugs are identified via manual investigation. Similarly, Beurdouche et al. systematically test TLS implementations for state machine bugs by checking whether an invalid protocol flow is accepted [38]. Invalid protocol flows are generated by skipping or repeating messages, or by hopping between message traces from different protocol modes. However, both approaches treat protocol messages as atomic units and do not challenge corresponding parsing routines.

TLSFuzzer [39] is a TLS test suite allowing to generate offensive TLS protocol flows with invalid messages. The expected implementation behaviour has to be specified in the test definitions explicitly; automated detection of incorrect implementation behavior does not seem to be supported.

Somorovsky presented a framework called *TLS-Attacker* [28], [40]. Besides testing for the susceptibility to known cryptographic attacks, it allows evaluating TLS server behavior by using a fuzzing approach based on dynamic modifications of message fields. *TLS-Attacker* detects implementation issues using instrumentation-based runtime monitoring of the target or by using a TLS context analyzer that investigates the correctness of TLS protocol flows.

Following the notion of differential testing [23], *SFADiff* is a generic approach based on *Symbolic Finite Automata* learning to systematically derive differences between a set of programs [41]. *SFADiff* has been evaluated successfully in three different settings, though none of them in the context of TLS. However, automata learning becomes prohibitively complex for TLS unless protocol interactions use a very limited alphabet.

Frankencerts [22] and *Mucerts* [24] apply differential testing to the certificate validation logic in TLS implementations. Where the latter disagree on the validity of one and the same randomly mutated X.509 certificate, a manual investigation of the root cause has to follow. However, neither approach addresses the TLS handshake itself.

Sivakorn et al. presented *HVLearn*, a black-box approach for testing hostname verification as part of TLS certificate validation [26]. From each implementation *HVLearn* infers a model that describes the set of all hostnames that match a given certificate template. Bugs in a model are detected by finding discrepancies with models of other implementations or by checking against regular-expression-based rules derived from the specification.

Petsios et al. presented *NEZHA*, an approach for domain-independent differential testing [25]. *NEZHA* uses the evolutionary fuzzing engine *libFuzzer* [42] guided by various metrics that specifically target differential testing. Applied to the certificate validation in TLS implementations, *NEZHA* finds significantly more discrepancies than *Frankencerts* and *Mucerts*.

We observe that in the context of TLS, differential testing has only been applied to the certificate validation (or parts thereof). Moreover do all previous approaches targeting TLS seem to involve developing dedicated source code that can handle the large variety of TLS message types. Not only does it significantly add to the complexity and the susceptability to bugs of the test tools themselves. It also makes the tools' maintenance laborious and binds the test quality more than necessary to the developer's thoroughness. In the following Section 4, we present an approach that allows a reduction of protocol-specific implementation efforts.

4 Generic Message Trees

Our test approach is based on interacting with TLS implementations via an exchange of TLS protocol messages. That is, at the very least we need to generate suitable and sufficiently diverse TLS messages for stimulation. The large diversity and complexity of TLS protocol messages poses a considerable challenge here. Moreover do we need to generate *invalid* messages just as we need to generate *valid* ones. These prerequisite can often only be met by dedicated and specifically implemented message handling routines.

For this purpose, we propose the concept of *Generic Message Trees* (GMTs). GMTs constitute an integral part of our methodology. In fact, it is the GMT concept that allows our methodology to be aware of the TLS protocol message syntax while being protocol-agnostic to the greatest possible extent.

4.1 The GMT Concept

The concept of GMTs is closely related to the concept of parse trees, but adds means e.g. for *in-place* message generation and manipulation. A GMT is an ordered rooted tree with typed nodes, which each represent a particular message component. A leaf node represents an atomic message field (e.g. an integer field) that allows a direct translation from and to its raw (*on-the-wire*) data representation. An internal node represents a composite message component whose content is recursively given by its child nodes.

Throughout this paper, we use the following notation. Let $\mathcal{T}$ denote the set of all finite GMTs (including the empty GMT $\varnothing$ with zero nodes), and, given some GMT $t \in \mathcal{T}$, let $\mathcal{V}_t$ denote the set of all nodes of t. Each node $v \in \mathcal{V}_t$ defines a subtree $\hat{v} \in \mathcal{T}$ that consists of node v and all its descendants. Furthermore, let $\mathcal{S}$ be the set of all finite octet (byte) strings. The length (number of octets) of an octet string $s \in \mathcal{S}$ is referred to as $|s|$.

As an example, Fig. 2 shows an excerpt of the GMT representation of a TLS ClientHello message.

4.2 Dissection and Serialization

Dissection denotes the process of translating the raw data representation $s \in \mathcal{S}$ of some message into its GMT representation $t \in \mathcal{T}$. Dissection can be expressed as a function $\mathbf{D} : \mathcal{S} \to \mathcal{T}$. The reverse operation is called *serialization* and is achieved by traversing the GMT depth-first and concatenating the raw data representation of each leaf node $v \in \mathcal{V}_t$. Serialization can be expressed as a function $\mathbf{S} : \mathcal{T} \to \mathcal{S}$.

Observe that by their definition, GMTs are an inherently generic concept. Most format-specific aspects are essentially abstracted by the dissection function $\mathbf{D}$. Therefore, $\mathbf{D}$ is highly dependent on the format definition of the message to be dissected. This is in contrast to the serialization function $\mathbf{S}$, which is fully generic.

4.3 Deriving Dissection Functions

Obviously, the benefits of GMTs were small if there was no efficient and automated way to obtain implementations of $\mathbf{D}$. Our approach to achieve this is based on the *TLS Presentation Language* (TPL). TPL is used more or less consistently throughout the TLS specification for the purpose of defining the content and encoding of TLS protocol messages [1]. TPL comprises basic message fields (i.e. integer and enumeration fields) as well as lists (also referred to as *vectors*) of fields, constructed (composite) types, and variants (i.e. dynamic choices within constructed types). Message definitions in TPL use a syntax quite close to that of the "C" programming language.

In the light of the aforementioned, TPL presents itself as an interesting candidate for automatically deriving implementations of $\mathbf{D}$. However, as TPL suffers from a somewhat casual and informal definition, for our purposes we use *eTPL*, an enhanced version of TPL [43]. eTPL features an improved expressiveness and is suitable for an automated generation of parsers, i.e. implementations of $\mathbf{D}$, and other format-specific routines. As a consequence, the task of manually implementing TLS-specific details can be minimized. It essentially boils down to copying the TPL-based message definitions from the TLS specification and using eTPL constructs to complement the definition by those features of the message format that TPL fails to cover.

4.4 GMT Operators

Given the generic nature of GMTs we define a *GMT operator* O as an abstract manipulation agent acting on a GMT t. An operator O is invoked on a certain GMT node $v \in \mathcal{V}_t$, but may affect any node in the GMT. For example, the node an operator is invoked on may merely serve as a starting point for sophisticated operations walking the whole GMT.

Not every operator is applicable to every node in a meaningful way. Thus, each operator O comes with a *filter function* $F_O : v \mapsto f \in \{0, 1\}$, which returns 1 if operator O may be invoked on node v, and 0 otherwise.

Observe that certain operations on a GMT or its nodes change the GMT in such a way that redissecting it (i.e. dissecting the tree's serialization) yields a structurally different GMT, i.e. $\mathbf{D}(\mathbf{S}(t)) \neq t$. Such *redissection instability* of a GMT is primarily caused by a violation of dependencies between message fields (e.g. out-of-bounds lengths) [1].

5 Methodology

As the integral component of our methodology for testing implementations of TLS – the handshake in particular – we design a fuzzing algorithm that is capable of generating

1. Redissection instability turns out to be a property worth considering when tracing the code path an independent parser is following while processing a GMT's serialization.

```
16 03 01 01 2C 01 00 01 28 03 03 40 C7 0E 24 30 01 B9 6D 8C 63 68 77 38 69 64 32 D3 E6 F9 49 10 7A AB AD 84 50 CD FF D6 A2 66 E4 00 00 92 C0 30 ...
--------------------------------------------------------------------------------------------------------------------------------------------------
{0:TLSRecord}                    TLSRecord
|--[0:type]                      RecordType                | 16                        | handshake(22)
|--[1:version]                   ProtocolVersion           | 03 01                     | TLSv1(769)
|--[2:length]                    Integer                   | 01 2C                     | 300
 --{3:msg}                       HandshakeMessage
   |--[0:type]                   HandshakeType             | 01                        | client_hello(1)
   |--[1:length]                 Integer                   | 00 01 28                  | 296
    --{2:msg}                    ClientHello
      |--[0:version]             ProtocolVersion           | 03 03                     | TLSv1_2(771)
      |--[1:random]              OpaqueBlob                | 40 C7 0E 24 30 01 B9 6D |
      |                                                    | 8C 63 68 77 38 69 64 32 |
      |                                                    | D3 E6 F9 49 10 7A AB AD |
      |                                                    | 84 50 CD FF D6 A2 66 E4 |
      |--[2:session_id_length]   Integer                   | 00                        | 0
      |--[3:session_id]          OpaqueBlob                |                           |
      |--{4:cipher_suites}       ClientHello_cipher_suites
      |  |--[0:_N]               Integer                   | 00 92                     | 146
      |   --{1:_V}               DynamicVector
      |     |--[0:CipherSuite]   CipherSuite               | C0 30                     | TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384(49200)
      ...   ...                  ...                         ...
```

Fig. 2. Illustration of the raw data representation (octet string in hexadecimal notation above the horizontal line) as well as the GMT representation (tree structure below the horizontal line) of the first part of a TLS *ClientHello* message. GMT nodes are printed in the same order as they are traversed for serialization. The four columns of the GMT's visualization reflect (from left to right) the tree structure, the nodes' types, the leaf nodes' raw data representations, and a human-readable representation (where applicable), respectively.

highly diverse corpuses of test messages. Our algorithm is designed such that the generated messages are very effective in provoking response discrepancies among different TLS implementations. In the spirit of differential testing, we then use such discrepancies as means to identify bugs in the implementations under test.

Actually, fuzz testing typically involves monitoring the test target for evidence of faulty behavior; for example using code instrumentation or dynamic memory monitoring. However, such measures may not be possible, e.g. because no source code is available. Moreover may some bugs, e.g. semantic bugs [44], not become manifest in the form of program crashes or invalid memory accesses. Therefore, combining differential testing with fuzz allows to detect less obvious failure cases.

With our methodology we try to close the gap between model-based approaches [23], [32], [33], [34], [35], [36], [37] – for which TLS is prohibitively complex, unless protocol interactions are significantly simplified – and approaches that are based on explicit test oracles [28] or hand-crafted protocol interactions [39]. In order to support settings that do not allow code instrumentation or dynamic memory monitoring, we intentionally refrain from using techniques in the design of our methodology that would break its black-box feature.

In the following, we use the term *implementation peer group*, or simply *peer group*, to refer to the set of protocol implementations that receive the same stimulation input and of which responses are compared among each other. *Test corpus* refers to a set of individual TLS messages which are used to stimulate implementations in the peer group.

5.1 Input Generation

Our fuzzing algorithm is based on the manipulation of template TLS messages. To this end, it makes use of the concept of GMTs and corresponding manipulation operators as introduced in the previous Section 4. In contrast to other black-box fuzzing approaches we are aware of, our approach allows to cover the space of mostly-valid TLS messages with minimal implementation effort, yet very high efficiency.

5.1.1 Manipulation Operators

We define the following set of *deterministic* operators. Where applicable, v' refers to node v after the operator application.

- **Voiding operator $\boldsymbol{O}_{\text{void}}$**: Replace subtree $\hat{v}$ by an empty leaf node. This operation effectively removes all contributions of $\hat{v}$ to the GMT's serialization, but retains a leaf node as a place holder potentially important for subsequent operations.
- **Removing operator $\boldsymbol{O}_{\text{rem}}$**: Fully remove subtree $\hat{v}$. The effect of this operator on the serialization of node v is similar to that of $\boldsymbol{O}_{\text{void}}$.
- **Duplicating operator $\boldsymbol{O}_{\text{dupl}}$**: Duplicate subtree $\hat{v}$ and insert the copy as a sibling, i.e. as a child of v's parent to the right of v.

Additionally, we define several fuzzing operators.

- **Truncating fuzz operator $\boldsymbol{O}^{\text{fuzz}}_{\text{trunc}}$**: Remove and truncate the nodes of subtree $\hat{v}$ such that its serialization after the operator's application is truncated to length n, i.e. $|\mathbf{S}(\hat{v}')| = n$. The length n after truncation is chosen uniformly at random between zero and $|\mathbf{S}(\hat{v})| - 1$.
- **Integer fuzz operator $\boldsymbol{O}^{\text{fuzz}}_{\text{int}}$**: Set the integer i represented by leaf node v to a new value i'. In *full range mode*, i' is chosen uniformly at random between zero and $2^w - 1$, where w is the integer field's bit width. In *proximity mode*, i' is chosen uniformly at random between zero and $2 \cdot i + 1$. Proximmity mode helps keeping integers close to typical values. For each application of $\boldsymbol{O}^{\text{fuzz}}_{\text{int}}$, full range mode or proximity mode are chosen at random with equal probabilities. This operator is only applicable to leaf nodes representing integers.

- **Content fuzz operator** $\boldsymbol{O}^{\text{fuzz}}_{\text{cont}}$: Randomize the raw data representation of leaf node v. The length is either left unchanged (*update-only mode*) or chosen uniformly at random between zero and $2|\mathbf{S}(\hat{v})| + 1$ (*resize-and-update mode*). For each application of $\boldsymbol{O}^{\text{fuzz}}_{\text{cont}}$, the choice between update-only mode and resize-and-update mode is made at random with equal probabilities. For $|\mathbf{S}(\hat{v})| = 0$ resize-and-update mode is chosen with a probability of one. This operator is only applicable to leaf nodes.
- **Appending fuzz operator** $\boldsymbol{O}^{\text{fuzz}}_{\text{app}}$: Insert a new leaf node with randomized content as a child of node v. The raw data length is chosen uniformly at random between one and four. This operator is only applicable to internal nodes.
- **Synthesizing fuzz operator** $\boldsymbol{O}^{\text{fuzz}}_{\text{syn}}$: Replace the subtree $\hat{v}$ by a semi-randomly synthesized subtree that obeys the message syntax inherent in the type of node v. Despite synthesizing *syntactically* correct message, $\boldsymbol{O}^{\text{fuzz}}_{\text{syn}}$ is not aware of the corresponding *semantics*.

Note that, conceptually, $\boldsymbol{O}^{\text{fuzz}}_{\text{syn}}$ is the most powerful operator. Based on the dissection functions derived from eTPL definitions and implanted in GMT nodes, it is capable of synthesizing message components that are not present in the template message. In this way, any TLS extension may be synthesized in a syntactically correct way.

5.1.2 *Resolving Inconsistencies (Repairing)*

The application of certain manipulation operators is likely to destroy the internal consistency of a GMT. For instance, when a variable-length message component is truncated, the corresponding length field should be updated.

Obviously, there is a trade-off between intentionally generating inconsistent messages (trying to challenge an implementation's input validation routines) and avoiding message inconsistency (trying to prevent an early rejection of invalid messages).

Therefore, we define a repair algorithm resolving inconsistency among the nodes of a GMT on a semi-random basis (see Fig. 3). The algorithm proceeds by starting at some node v and traversing the GMT towards its root. For each node on this path, local inconsistencies among the node's direct children are resolved using a node-specific repair routine[2]. If run in randomized mode, nodes are skipped randomly with a probability of $p = 1/2$, giving rise to incidental consistency violations within the GMT.

5.1.3 *Input Generation Algorithm*

The core of our algorithm for TLS message generation is presented in Fig. 4. It starts from the GMT representation of a template message and repeatedly applies a randomly chosen manipulation operator to a suitable, randomly selected GMT node. After each operator application (excluding $\boldsymbol{O}^{\text{fuzz}}_{\text{int}}$), the repair algorithm is invoked on the node to which the previous operator has been applied. The number of operator applications is implicitly chosen at random; after each iteration the algorithm stops with a probability of $p = 1/2$. The serialization of the final GMT is the message to be used for stimulation[3].

```
1: procedure REPAIR($v$, randomize)
2:     while $v \neq \bot$ do                         ▷ Repeat until done with root
3:         if not randomize or RND($\{$true, false$\}$) then
4:             REPAIRLOCAL($v$)
5:         end if
6:         $v \leftarrow$ PARENTOF($v$)                 ▷ One step towards root
7:     end while
8: end procedure
```

Fig. 3. The repair algorithm used for restoring consistency between nodes within a GMT. Starting from node v, the algorithm moves towards the GMT's root. In deterministic mode ($\text{randomize} = \text{false}$), for each node on the path towards the root local inconsistency among each node's direct children are resolved using a node-specific function REPAIRLOCAL. In randomized mode ($\text{randomize} = \text{true}$), nodes are skipped randomly with a probability of one half (RND function call). RND($\mathbf{X}$) returns an element $x \in \mathbf{X}$ uniformly drawn at random from the set $\mathbf{X}$. PARENTOF(v) returns the parent node of v or $\bot$ if v is the root node.

In order to avoid the generation of bit-identical duplicates, we keep a list of the hashes of previously generated messages. Having said that, this list is the only state our generation algorithm holds.

Unfortunately, unless recording internet traffic at large scale, obtaining a large corpus of TLS template messages is a nontrivial problem. As we show in Section 6, our algorithm turns out to be effective even if only a single template messages is provided.

5.2 Response Analysis

In the following, we describe the differential analysis of responses received from the implementations in the peer group upon stimulation.

5.2.1 *Response Reduction*

As a consequence of testing TLS protocol implementations, the responses we have to deal with are complex TLS protocol messages that lack straightforward comparability. For differential testing, it is therefore crucial to define a reasonable metric that can assess whether responses are discrepant or in agreement – potentially despite differences in their raw data representations. We abstract the concrete choice of this metric by using a *reduction function* R that maps a TLS implementation's response, represented as a GMT $t \in \mathcal{T}$, to a *reduced response set* $\mathcal{R}$, i.e. $R : \mathcal{T} \rightarrow \mathcal{R}$. Responses $t_A, t_B \in \mathcal{T}$ from two implementations A and B are considered in agreement under reduction function R if $R(t_A) = R(t_B)$ and discrepant otherwise[4].

In the following, let $\mathcal{P}$ denote the implementation peer group and let $\mathcal{X}$ refer to a test corpus, i.e. a set of stimuli $x \in \mathcal{X}$. Furthermore, let $t_{i,x}$ denote the GMT representation

2. These repair routines are automatically derived from the eTPL-based message definitions just as the dissection functions are.

3. Note that most protocol-specific dependencies of our algorithm are encapsulated in the corresponding GMT dissection and repair routines.

4. It is clear that the choice of a reduction function R affects both the sensitivity to real implementation issues as well as the susceptibility to innocuous discrepancies. Optimizing the choice of R is out of the scope of this paper though.

```
1:  function FUZZ($v_0$)                                          ▷ Fuzz $\hat{v_0}$, the subtree whose root is $v_0$
2:      repeat
3:          $o \leftarrow$ RND($\{O_{\text{void}}, O_{\text{rem}}, O_{\text{dupl}}, O^{\text{fuzz}}_{\text{trunc}}, O^{\text{fuzz}}_{\text{int}}, O^{\text{fuzz}}_{\text{cont}}, O^{\text{fuzz}}_{\text{app}}, O^{\text{fuzz}}_{\text{syn}}\}$)    ▷ Select an operator at random
4:          $v \leftarrow$ RND($\{v \in \mathcal{V}_{\hat{v_0}} | F_o(v) = 1\}$)    ▷ Select a suitable node in $\hat{v_0}$ at random
5:          APPLYOPERATOR($o, v$)                                 ▷ apply operator $o$ to node $v$
6:          if $o \neq O^{\text{fuzz}}_{\text{int}}$ then          ▷ Unless an integer field has been fuzzed ...
7:              randomize $\leftarrow$ RND($\{$true, false$\}$)
8:              REPAIR($v$, randomize)                            ▷ ... repair GMT (e.g. update length fields)
9:          end if
10:         if $o \in \{O_{\text{dupl}}, O^{\text{fuzz}}_{\text{syn}}\}$ and RND($\{$true, false$\}$) then    ▷ Duplicated or synthesised subtrees ...
11:             FUZZ($v$)                                         ▷ ... may be fuzzed recursively
12:         end if
13:     until RND($\{$true, false$\}$)     ▷ The number of operator applications becomes approx. exponentially distributed
14:     return $v_0$                                              ▷ Return the manipulated (sub)tree
15: end function
```

Fig. 4. The fuzzing algorithm used to semi-randomly generate mostly-valid TLS messages. The algorithm starts from the GMT representation v_0 of a template message and repeatedly applies a randomly selected manipulation operator to a randomly selected node. In-between operator applications message inconsistencies are resolved on a semi-random basis using the repair algorithm shown in Fig. 3. RND($\mathbf{X}$) returns an element $x \in \mathbf{X}$ uniformly drawn at random from the set $\mathbf{X}$.

of the response obtained from implementation $i \in \mathcal{P}$ when stimulated with stimulus $x \in \mathcal{X}$. We define the *response signature* as the tuple of implementation responses reduced under some R as

$$T_R(\mathcal{P}, x) = \left\langle R(t_{1,x}), \ldots, R(t_{|\mathcal{P}|,x}) \right\rangle . \quad (1)$$

Clearly, we are most interested in cases where a *single* stimulus induces *discrepant* responses in the peer group. As an indicative measure, we define the number of *unique responses* obtained for a single stimulus as

$$N_R(\mathcal{P}, x) = \left|\left\{ R(t_{i,x}) \mid i \in \mathcal{P} \right\}\right| . \quad (2)$$

Obviously, $1 \leq N_R(\mathcal{P}, x) \leq |\mathcal{P}|$ with $N_R(\mathcal{P}, x) = 1$ if[5] and only if all implementations in $\mathcal{P}$ are in agreement with respect to R.

5.2.2 Identification of Root Causes

Once a response discrepancy has been detected, determining its root cause is a nontrivial problem. For the time being this remains as a manual task for the human tester. We sketch some ideas on how to address this challenge in Section 8 about future work.

Note that behavorial discrepancies may not only be due to implementation bugs. As alternative causes dissimilar features or configurations of the implementations as well as different interpretations of the specification need to be considered before attributing discrepant implementation behavior to a bug.

5.3 Capabilities and Limitations

In principle, our approach allows detecting any implementation issue that gets provoked by the stimulation input and then become manifest in an implementation's response behavior. That is, it does not require an implementation to crash or to perform an invalid memory access. Assuming the reduction function does not "map away" these deviations, an observable deviation of an implementation's output from those of the other implementations is sufficient. It is this fact that yields sensitivity to semantic bugs which might otherwise elude detection.

On the other hand, differential testing is generally blind with respect to defects that are present in all implementations in the peer group or that result in identical or indistinguishable (mis)behavior. Furthermore, silent issues that never affect the implementation's response behavior (e.g. memory leaks) stay unnoticed, unless the stimulation provoking them lets other implementations behave observably different.

Currently, interactions with the TLS implementations under test are given by distinct and reiterate stimulus-response pairs. Our test system acts as a TLS client by repeatedly sending semi-randomly generated ClientHello messages over fresh transport connections (stimulus) and observing the TLS servers' responses. While this constitutes an obvious restriction (it only tests the ClientHello processing routines within TLS servers), we see our approach as presented herein only as the first step towards *fully interactive* differential testing of black-box TLS protocol implementations, covering both servers and clients as well as the full TLS handshake process. Dealing with the stateful nature of TLS and its use of cryptography constitutes the major challenge in that direction (see also Section 8 on prospects and future work).

6 EVALUATION

In this and the following Section 7 we report on our results obtained from applying our test approach

5. We deliberately ignore the somewhat degenerate case of an empty peer group here.

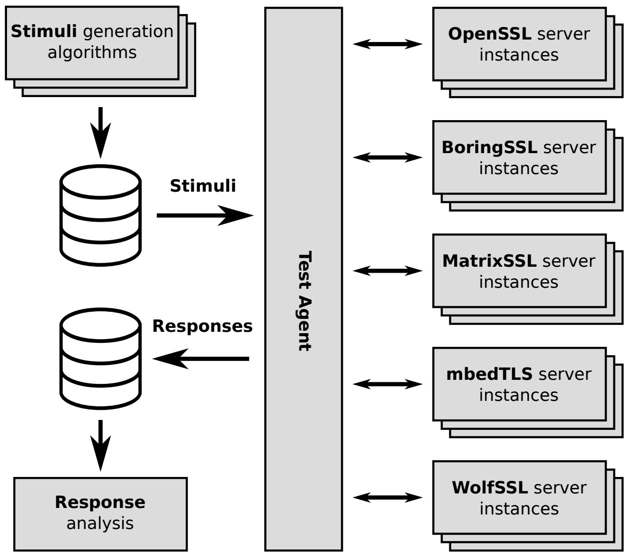


Fig. 5. Illustration of the experimental setup that we use to compare our test approach against AFL, *TLS-Attacker*, and *NEZHA*. Stimuli from different pre-generated test corpuses are used to stimulate five different TLS server implementations via their standard network interfaces. For each TLS implementation we launch 20 independent instances in order to speed up the test process through parallelization. Implementation responses are recorded for further analyses, but do not feed back to input stimulation. All components run on a single local machine.

to ***OpenSSL 1.0.2h***, ***BoringSSL***[6], ***MatrixSSL 3.8.4***, ***mbedTLS 2.2.0***, and ***WolfSSL 3.9.8***. That is, our peer group $\mathcal{P}$ is given by these five implemenations. The list includes implementations for both PC as well as embedded platforms. We refrain from using more than five implementations so as to keep the complexity of investigating response discrepancies at a manageable level[7].

In order to evaluate the efficiency and effectiveness of our generation algorithm, we compare our approach to other methods, i.e. ***American Fuzzy Lop*** (AFL) [27], ***TLS-Attacker*** [28], and ***NEZHA*** [25].

6.1 Experimental Setup

Our experimental setup is sketched in Fig. 5. Using pre-generated test corpuses, we stimulate each TLS server implementation in our peer group and record corresponding responses for further analyses (see Section 6.1.2). Implementation behavior does not feed back to input stimulation. Additionally, we determine the code coverage in *MatrixSSL 3.8.4* using *gcov* version *4.8.2* for each test corpus as a whole.

All components of our setup run on a single local machine (standard office PC, *Linux*-3.11-2 64 bit, Intel Core i5-4590, 8 GB RAM). TLS server implementations are compiled with *gcc* version 4.8.2 and use a similar compile and runtime configuration. The runtime configuration includes one and the same X.509 RSA server certificate and TLS 1.2 as the highest supported protocol version.

Communication with the TLS server implementations proceeds via the loopback network interface. To speed up the process, we parallelize test interactions by launching 20 independent instances of each server implemenation in the peer group. In this way, our setup manages to handle on average approx. 55 stimuli (i.e. 55×5 server stimulations) per second using a per-instance response timeout of 100 ms.

6.1.1 Input Generation

For each generation method, i.e. our algorithm, AFL, *TLS-Attacker*, and *NEZHA* we build 100 independent test corpuses with 100'000 stimuli each.

As a start we use our generation algorithm described in Section 5.1.3 to generate a corresponding set of test corpuses. As template message we use one single ClientHello message with a length of 305 octets generated by *OpenSSL*'s command-line client (version *1.0.2h*). The template ClientHello message offers TLS 1.2 as highest supported protocol version and lists 72 supported cipher suites. Furthermore, it contains five different extensions: *Supported point formats* (with three entries) and *Supported Elliptic Curves* (with 25 entries) [46], *Supported signature algorithms* (with 15 entries), *Session ticket* (empty) [47], and *Heartbeat* [48]. Given these settings, our algorithm generates approx. 500 TLS messages per second.

Additionally, we use AFL version *2.40b* run against an appropriately instrumented instance of *MatrixSSL 3.8.4* to generate a similar set of independent **AFL test corpuses**. AFL is provided with the same single ClientHello message that we also use to seed our generation algorithm. We let AFL skip determinstic generation steps as we expect those to introduce a bias in the test corpus[8].

Using the *TLS-Attacker* framework [28] we build a set of ***TLS-Attacker* test corpuses** based on the ClientHello messages emitted by *TLS-Attacker*. We run a slightly modified[9] version of *TLS-Attacker* against *OpenSSL 1.0.2h*. Note that *TLS-Attacker* generates a large number of TLS protocol flows where ClientHello messages differ only in the *ClientRandom* field. In order to allow for a fair comparison, we therefor drop those quasi-duplicates before building a test corpus.

Finally, we let *NEZHA* [25] run against the server implementations in our peer group to generate a set of ***NEZHA* test corpuses**. We let *NEZHA* be guided by its output δ-diversity metric using distinct return codes for a ServerHello response, each different type of Alert message, and no response at all within a timeout of 100 ms[10].

Note that, in contrast to our approach, AFL, *TLS-Attacker*, and *NEZHA* use life interactions with one or more TLS implementations to guide the input generation process. Nevertheless does our evaluation in all cases use the setup

6. The version we use corresponds to commit `78684e5b222645828ca302e56b40b9daff2b2d27` on branch `2883` of the official *BoringSSL* Git repository [45].

7. The complexity of telling apart interesting from benign response discrepancies suffers from scaling unfavorably with the size of the peer group $\mathcal{P}$. Addressing this scalability issue in the context of TLS is going to be the subject of future work.

8. AFL would run much more than 100'000 determinstic generation steps.

9. We added a few lines in the framework's source code that allow recording the generated ClientHello messages on disk. Our version is based on commit `fd74472c9950c4415a88934a3c21808ac3b078d3` from *TLS-Attacker*'s official GitHub repository [40].

10. This reduction scheme is equivalent to the reduction function R_2 defined in Eq. (5) of the following Section 6.1.2.

shown in Fig. 5 to consistently record implementation responses based on previously generated test corpuses.

6.1.2 Evaluation Metrics

As our primary evaluation metric, we define the number of *unique response discrepancies* Δ_R as

$$\Delta_R(\mathcal{P}, \mathcal{X}) = \left|\left\{T_R(\mathcal{P}, x) \mid x \in \mathcal{X} : N_R(\mathcal{P}, x) > 1\right\}\right| \quad (3)$$

with $T_R(\mathcal{P}, x)$ as defined in Eq. (1) and $N_R(\mathcal{P}, x)$ as defined in Eq. (2). $\Delta_R(\mathcal{P}, \mathcal{X})$ is a measure of the capability of test corpus $\mathcal{X}$ to induce distinct discrepancies among the peer group $\mathcal{P}$'s implementations[11].

Actually, we are not only interested in the *total* number of unique response discrepancies induced by a test corpus $\mathcal{X}$ *as a whole*. Rather, we are mostly interested in *cumulative* numbers obtained from gradually increasing the number of stimuli taken from the test corpus. Therefore, by $\mathcal{X}|_n$ we denote a *truncated* test corpus that only consist of those n stimuli $x \in \mathcal{X}$ that have been generated first. We then use $\Delta_R(\mathcal{P}, \mathcal{X}|_n)$ as a function of n with $0 \leq n \leq |\mathcal{X}|$.

In accordance with Section 5.2.1 we define two response reduction functions $R_{1,2} : \mathcal{T} \rightarrow \mathcal{R}_{1,2}$ that feature different output granularity. R_1 has a binary outcome with $\mathcal{R}_1 = \{\text{“SH”}, \text{“Abort”}\}$. It reflects whether or not a server responds with a ServerHello message within a timeout of 100 ms, i.e.

$$R_1(t) := \begin{cases} \text{“SH”} & \text{if } t \equiv \text{ ServerHello} \\ \text{“Abort”} & \text{otherwise.} \end{cases} \quad (4)$$

R_1 is particularly suitable to find those stimuli where all implementations except one refuse to continue the handshake. In such cases, the deviant implementation is most likely to miss an important input validation check.

As a more fine-granular option, R_2 additionally takes into account error information in case no ServerHello message is received within the 100 ms timeout. With

$$\mathcal{R}_2 = \{\text{“SH”}, \text{“AL/0”}, \ldots, \text{“AL/255”}, \text{“E”}\}$$

we set

$$R_2(t) := \begin{cases} \text{“SH”} & \text{if } t \equiv \text{ ServerHello} \\ \text{“AL/}x\text{”} & \text{if } t \equiv \text{ Alert of type } x \\ \text{“E”} & \text{if } t = \varnothing \text{ (empty).} \end{cases} \quad (5)$$

Note that $R_2(t) = \text{“E”}$ either if a server silently closes the connection on TCP level or if it does not reply at all within the 100 ms timeout.

As a second evaluation metric, we use the code (line) coverage determined in *MatrixSSL 3.8.4*. For each test corpus, the code coverage is determined over stimuli from the test corpus as a whole. Note that we restrict the coverage analysis to those modules in *MatrixSSL* that are responsible for parsing incoming TLS messages[12].

11. Note that if the condition $N_R(\mathcal{P}, x) > 1$ were dropped, the definition of Δ_R would be identical to the *output δ-diversity* defined by Petsios et al. [25].

12. Concretely, these are `sslDecode.c` (parsing TLS records), `hsDecode.c` (parsing handshake messages), and `extDecode.c` (parsing ClientHello and ServerHello extensions) from the *MatrixSSL* sources. The code considered amounts to 1617 executable lines in total.

TABLE 1
Average code (line) coverage determined in *MatrixSSL 3.8.4*

Test approach	Average line coverage [%]
Our approach	26.6 ± 0.3
AFL	25.7 ± 0.4
NEZHA	24.0 ± 0.6
TLS-Attacker	21.9 ± 0.2

6.2 Results

In this section we present the result of our evaluation using the metrics presented in the previous Section 6.1.2.

6.2.1 Unique Response Discrepancies

Fig. 6 shows the average of $\Delta_R(\mathcal{P}, \mathcal{X}|_n)$ as defined in Eq. (3) and its standard deviation plotted individually for our approach, AFL, *TLS-Attacker*, and *NEZHA* as well as the two reduction functions $R = R_1$ and $R = R_2$ defined in Eqs. (4) and (5), respectively. For each generation method, the average is computed over the respective 100 test corpuses.

It turns out that our generation algorithm is more effective in provoking discrepant resonses than AFL, *TLS-Attacker*, and *NEZHA* are. Considering test corpuses as a whole (untruncated), our approach on average generates 17 % more (for $R = R_1$) and 76 % more (for $R = R_2$) unique response discrepancies than the respective second most effective approach.

The advantage of our approach seems to be present at least for test corpuses not significantly larger than 100′000 stimuli. However, recall that our approach basically is stateless. Δ_R is therefore likely to run into saturation at some point. In contrast to that follow the other approaches either an evolutionary (AFL and *NEZHA*) or a partially deterministic strategy (*TLS-Attacker*), making the long-term behavior hard to predict. An illustrative example is the abrupt jump in Δ_R at $n \approx 30'000$ for *TLS-Attacker*.

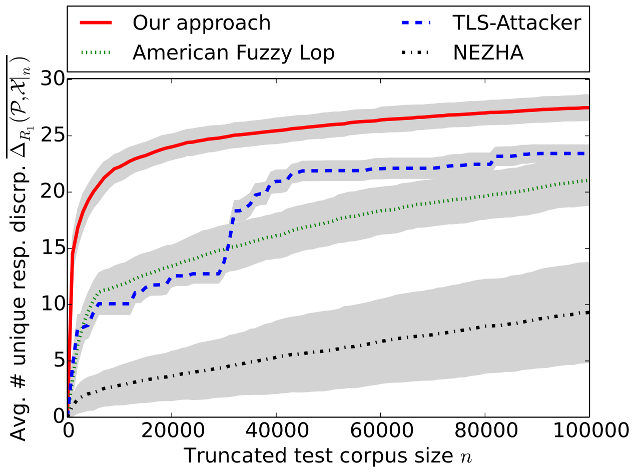
Furthermore, it is a notable observation that in our study AFL clearly outperforms *NEZHA*. This is in contrast to what the authors of *NEZHA* found when applying *NEZHA* to the validation routines of X.509 certificates in TLS implementations [25]. We surmise that our observation is due to the fact that we use only one single TLS message to seed the generation process, whereas in the case of certificate validation AFL and *NEZHA* have been given 1000 template certificates.

6.2.2 Code Coverage

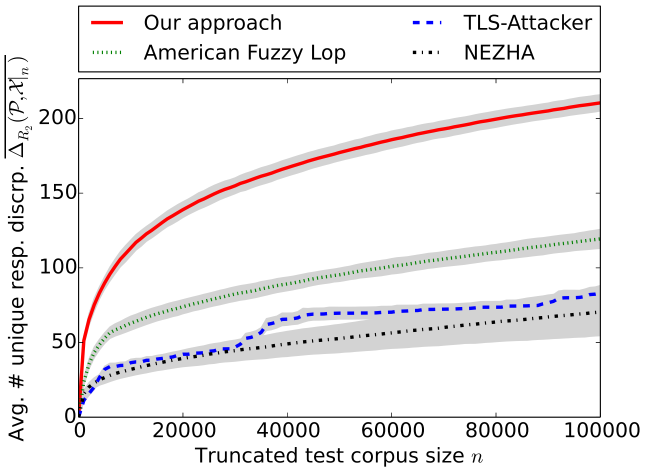
Just as for Δ_R, the average code (line) coverage in *MatrixSSL 3.8.4* and its standard deviation is computed over the 100 test corpuses of each generation method individually.

The resulting numbers are given in Table 1. Reaching approx. 26.6 % of the executable lines in the corresponding modules of *MatrixSSL 3.8.4*, test corpuses generated with our algorithm achieve a code coverage slightly higher than that achieved with any other evaluated approach (≤ 25.7 %).

Recall that our evaluation is restricted to server stimulations via ClientHello messages. The fact that the achieved



(a) Reduction to $\mathcal{R}_1 = \{\text{“SH”}, \text{“Abort”}\}$



(b) Reduction to $\mathcal{R}_2 = \{\text{“SH”}, \text{“AL/0”}, \dots, \text{“AL/255”}, \text{“E”}\}$

Fig. 6. The average number of unique response discrepancies $\overline{\Delta_R(\mathcal{P}, \mathcal{X}|_n)}$ for Δ_R as defined in Eq. (3), plotted as a function of the truncated test corpus size n. $\overline{\Delta_R}$ is plotted for test corpuses generated with our approach (red solid line), AFL (green dotted line), *TLS-Attacker* (blue dashed line), and *NEZHA* (black dash-dotted line). The average is taken for each generation method individually over the respective 100 test corpuses; the grey bands represent the corresponding standard deviation. The plot is given for the two reduction functions $R = R_1$ (left) and $R = R_2$ (right) defined in Eqs. (4) and (5), respectively.

code coverage is considerably below 100 % is due to most of the unreached lines in *MatrixSSL* being responsible for parsing incoming messages other than ClientHello messages.

7 A Study of Some Identified Bugs

In the following, we briefly study some bugs in *MatrixSSL 3.8.4* and *WolfSSL 3.9.8* that we revealed with the help of our test approach. Among these bugs are two more or less independent ones in *MatrixSSL 3.8.4* that in combination we consider as a serious vulnerability.

In total, we identified 16 issues attributable to distinct root causes with varying impact on security or interoperability. We refrain from listing every single issue but discuss the essential aspects of our findings in Section 7.3. In all cases did we responsibly disclose our findings to the corresponding developers. In the meanwhile fixes for all serious and some minor bugs have been released.

For the rest of this section we drop the version numbers and simply use *MatrixSSL* and *WolfSSL* to refer to *MatrixSSL 3.8.4* and *WolfSSL 3.9.8*, respectively.

7.1 Mishandling of Length Fields

A particularly interesting – and apparently very frequent – class of issues in TLS implementations is related to an incorrect or inconsistent handling of the length fields of nested message components during parsing. As a result, the faulty parser may misinterpret the message content or may even be vulnerable to attacks.

Fig. 7 illustrates situations that can expose such parser misbehavior. In a well-formed message the length of a message component (the outer frame) is consistent with the lengths of its nested components (the inner frames). An *overflow* situation with *excess data* arises if the length of the outer frame exceeds the total length of its inner frames. In contrast to that does an *underflow* situation with *deficit data* arise if the total length of the inner frames exceeds the length of the outer frame. Explicitly designed to do so, our algorithm presented in Section 5.1 generates messages that randomly exhibit overflows and underflows situations in various message parts.

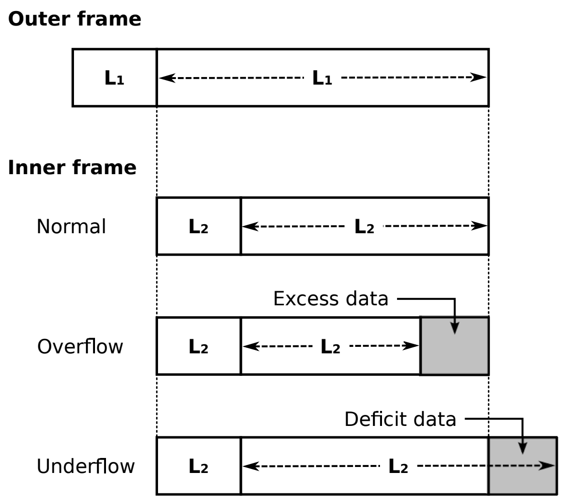


Fig. 7. Illustration of situations that may expose a message parser's mishandling of the length fields of nested message components. While normally redundant length fields are consistent, inconsistencies result in an *overflow* situation with real *excess data* (data provided by the outer frame of which the inner frame cannot make sense) or an *underflow* situation with *deficit data* (data the inner frame expects but the outer frame does not provide). In the absence of message fragmentation neither an overflow nor an underflow situation is expected or even acceptable.

In the following sections, we briefly report on two corresponding bugs we found in *MatrixSSL* and *WolfSSL*.

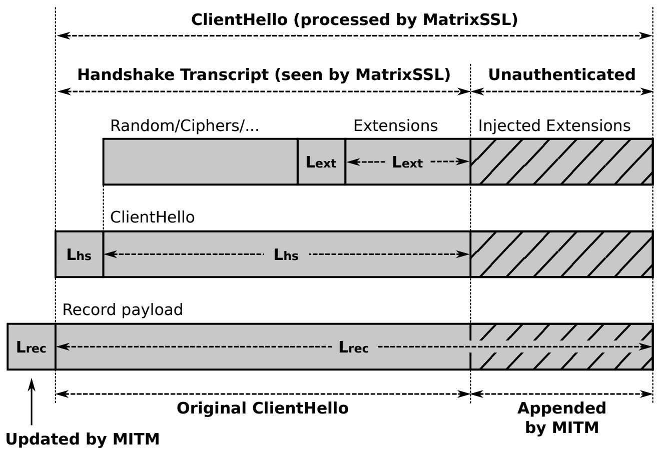


Fig. 8. Illustration of the vulnerability in *MatrixSSL* 3.8.4 that allows a man-in-the-middle (MITM) attacker to inject unauthenticated ClientHello extensions into a handshake between a *MatrixSSL* server and an arbitrary TLS client without either side noticing. The attacker simply appends data to the TLS record and only updates the length in the record's header. The vulnerability is a result of an incorrect/inconsistent treatment of length fields in the ClientHello processing routines of *MatrixSSL* 3.8.4. It allows to make the server process an extended ClientHello message while only the original ClientHello message contributes to the handshake transcript.

7.1.1 MatrixSSL: Unauthenticated ClientHello Extensions

Because of its insufficient adherence to and inconsistent treatment of length fields when parsing ClientHello messages, *MatrixSSL* allows a *man-in-the-middle* (MITM) an unnoticed injection of ClientHello extensions into the TLS handshake between a *MatrixSSL* server and an arbitrary TLS client.

Concretely, on the one hand, *MatrixSSL* does not use the length specified in the header of an incoming handshake message to restrict the processing of a ClientHello message, but instead simply assumes that the end of the ClientHello message is aligned with the end of the record. On the other hand, *MatrixSSL* ignores the field specifying the length of the ClientHello extension list, assuming that the end of the extension list is aligned with the apparent end of the ClientHello message[13].

Having said that, *MatrixSSL* *does* use the handshake header's length field to restrict what contributes to the handshake transcript. As illustrated in Fig. 8, this opens up an opportunity to a MITM to inject unauthenticated ClientHello extensions into any handshake with a *MatrixSSL* server by simply appending data to the TLS record containing the ClientHello message and only updating the record's length field.

Granting such power to a third party doubtlessly violates the assumptions that the security guarantees of TLS are based on. However, the severity of this vulnerability heavily depends on the capabilities of the injected extensions. Considering the list of extensions supported by *MatrixSSL*, we currently do not see an obvious way to exploit this vulnerability. This assessment is *not* based on a thorough analysis, though.

13. Note that because of the first issue, *MatrixSSL* effectively reads ClientHello extensions up to the end of the TLS record.

7.1.2 WolfSSL: Erroneous Processing of Excess Data

Similarly to *MatrixSSL*, *WolfSSL* does not fully respect the length of a ClientHello message as specified in the handshake header. When parsing a ClientHello message, excess data within the ClientHello message (i.e. data following the ClientHello extension list) is treated as the beginning of a subsequent handshake message. Though this bug in *WolfSSL* does not seem to induce a security issue, it certainly is in conflict with the TLS specification.

Note the difference to the corresponding issue in *MatrixSSL*. While *WolfSSL* treats excess data as if it belongs to a *lower* level of nesting (the record layer), *MatrixSSL* treats excess data as if it belongs to a *higher* level of nesting (the extension list).

7.2 Faulty Parameter Negotiation

In the course of a TLS handshake, a client typically proposes a list of protocol parameter options to the server, from which the server then makes a choice of its preference. Lacking a mutually supported option, the server actively aborts the handshake by sending an appropriate alert message. This mechanism applies e.g. to cipher suites, compression methods, and other parameters. Note that, while still selected dynamically, the protocol version is negotiated in a different way [14].

We give two examples of flawed negotiation implementations in the following.

7.2.1 MatrixSSL: Protocol Version Downgrade

Under certain circumstances, a *MatrixSSL* server does not select the highest TLS protocol version available between the client and the server. If the protocol version field in a ClientHello message (the field which informs the server about the highest version the client supports) is set to a version beyond TLS 1.2 (e.g. `0x0304` corresponding to TLS 1.3), the *MatrixSSL* server selects TLS 1.1, even though TLS 1.2 is supported by both client and server.

The security impact of this deviation from the specification currently seems negligible.[15] However, it might become relevant when TLS 1.2 turns out to be insecure in a way that an attacker can affect the handshake between legitimate parties.

7.2.2 Nonobservance of Supported Compression Methods

The server implementation of *MatrixSSL*, *WolfSSL*, and *mbedTLS 2.2.0* do not consider the list of compression methods offered by the client in the ClientHello message when making the corresponding choice. Rather, these implementations select the *no compression* option even if not listed by the client.

There is no impact on security resulting from this (mis)behavior.

14. Actually, with TLS 1.3 negotiation of the protocol version is going to be harmonized with the usual negotiation mechanism as described before [31].

15. TLS specification states that "*If a TLS server receives a ClientHello containing a version number greater than the highest version supported by the server, it MUST reply according to the highest version supported by the server*" [1].

7.3 Discussion

As indicated before, the issues described in the previous sections are not the only we found.

For instance, there are several further cases where TLS implementations fail to systematically validate the length fields of nested message components. This class of implementation flaws therefore presents itself as a quite common one. To some extent we attribute this to the fact that only the draft version of the upcoming TLS 1.3 explicitly addresses these issues in a general statement[16]. Previously released specifications are fairly faint in that respect as the general case is somewhat inauspiciously embedded in a very specific statement on ClientHello parsing[17].

Presumably, further issues we found are due to the TLS specification not being very explicit in certain respects, too. Generally, it tends to be vague with regard to a receiver's liability to enforce rules it imposes on senders. This effectively forces each individual implementor to balance the pursuits of robustness, security, and interoperability on his or her own[18]. Moreover, in some cases the specification does not determine default parameters to be used when parameter negotiation as part of the protocol is incomplete. Fortunately, the current draft specification of the upcoming version 1.3 of TLS eliminates at least some of these deficiencies [31].

Note that, even though the immediate security impact of certain implementation misbehavior might be negligible, it can legitimately be considered undesirable from a meticulous point of view. In principle even subtle and seemingly benign misbehavior might eventually by turned into an attack-driving lever if used elaborately by a smart attacker [18], [53].

8 Prospects and Future Work

Our findings seem to indicate that differential testing applied to the TLS handshake protocol is a promising direction for improving the quality of TLS implementations. That said, several limitations of our current approach should and are going to be addressed in future work.

8.1 Fully Interactive Differential Testing

We consider the realization of fully interactive differential testing of the TLS handshake as the most interesting direction for future work. It would allow going beyond merely testing ClientHello processing in server implementations. However, the stateful nature of TLS and its use of cryptography present themselves as the major difficulties towards that. Nevertheless do we believe that harmonizing these conflicting aspects is possible.

The key idea is to introduce a TLS-specific and stateful *crypto filter* in the stimuli generation and response analysis loop. While the GMT representations of outgoing and incoming messages get passed through the crypto filter, its state (i.e. cryptograhic algorithm selection, dynamically generated keys, etc.) is updated accordingly and cryptographic message components are inserted or removed, respectively. Thereby, the crypto filter essentially becomes the crypographic TLS endpoint and keeps stateful aspects and cryptographic details away from the core algorithm. In order to allow for message manipulations before and after cryptographic computations, messages might be passed back and forth between the generation algorithm and the crypto filter. The efficiency of such a setting remains to be studied though.

With fully interactive testing comes a break-up of the current separation between stimuli generation and implementation stimulation. While removing the advantageous parallelizability feature of our current approach, it would allow for response-driven feedback to guide and dynamically optimize the generation algorithm (e.g. similar to the output δ-diversity guidance of *NEZHA*). Bringing in machine learning techniques at this point might be interesting as well.

8.2 Further Research Directions

In addition to the aforementioned high-priority goal, there are further directions into which we plan to extend our work.

It is desirable to add support for further protocol versions, e.g. TLS 1.3, the upcoming successor of TLS 1.2 [31], and *Datagram TLS* (DTLS), the datagram version of TLS [54]. Moreover do we plan to add support for message fragmentation and coalescence at the TLS record layer.

Machine learning techniques might allow to simplify the task of manually correlating response discrepancies with root causes in the implementations. For instance, we plan to automatically derive specific characteristics in the stimulation messages that induce certain discrepancy patterns. In this way, the range of possible root causes might be narrowed down. Delta-debugging is another interesting approach in this direction [55].

Furthermore, as indicated previously, the long-term behaviour (i.e. for large test corpuses) of our approach should be studied and contrasted against that of others.

Finally, in the light of the high number of unique response discrepancies triggered by our stimulation messages, we expect our approach to be suitable for fingerprinting unknown TLS implementations [56]. We plan to study this potential in more detail.

9 Conclusion

In this paper, we presented our work on applying differential testing to black-box implementations of the TLS

16. The general statement is: "*Peers which receive a message which cannot be parsed according to the syntax (e.g., have a length extending beyond the message boundary or contain an out-of-range length) MUST terminate the connection with a "`decode_error`" alert.*" [31].

17. Here, the TLS specification states: "*A server MUST accept ClientHello messages both with and without the extensions field, and (as for all other messages) it MUST check that the amount of data in the message precisely matches one of these formats; if not, then it MUST send a fatal "`decode_error`" alert*" [1]. Note that being very strict with regards to such validation checks is in a way in conflict with Postel's Law which demands: "*Be conservative in what you do, be liberal in what you accept from others*" [49]. However, in addition to the debatable applicability of Postel's Law to security protocols [50], [51], allowing superfluous excess data in protocol messages unnecessarily provides an attacker with extra means for affecting the handshake transcript and exploiting transcript collisions [28], [52].

18. As a potentially positive consequence, however, it enables implementation-specific optimizations where certain validation checks might be dropped for the benefit of reduced resource demands.

protocol. In contrast to previous work using differential testing in this context, we do not focus on the certificate validation logic of TLS implementations but directly address the TLS handshake protocol itself.

To this end, we presented a novel fuzzing strategy for efficiently generating *mostly-valid* TLS protocol messages. Our algorithm turns out to be highly effective: in our peer group with *OpenSSL*, *BoringSSL*, *MatrixSSL*, *mbedTLS*, and *WolfSSL*, it induces more unique response discrepancies than American Fuzzy Lop, *TLS-Attacker*, and *NEZHA* do. Moreover does it achieve the highest code coverage in *MatrixSSL* among these alternative methdos.

Compared to standard fuzzing, the differential test approach can achieve sensitivity to implementation bugs that do not express themselves in program crashes or invalid memory access. It allowed us to reveal a number of issues in popular TLS implementations. Among these is a serious vulnerability in *MatrixSSL 3.8.4* which a man-in-the-middle can use to inject unauthenticated ClientHello extensions into the TLS handshake between a *MatrixSSL* server and an arbitrary client. In all cases did we disclose our findings responsibly to the respective developers; fixes are available in the meanwhile.

While currently we use fuzz-generated ClientHello messages to stimulate TLS servers, we see our approach as presented herein only as the first step towards *fully interactive* differential testing of black-box TLS protocol implementations. In this context, dealing with the stateful nature of TLS and its use of cryptography is an obvious challenge. However, our ongoing work indicates that this problem is solvable.

Anyway, our findings let us think that applying differential testing to the TLS handshake is a promising direction for identifying bugs in TLS implementations. Our approach may serve as a valuable complement to existing test tools for TLS. In particular, it may allow developers of low-footprint, C-based TLS implementations to benefit from the promise of correctness related to verified implementations like *miTLS* [21] written in high-level languages.

Our software tools are available[19] as open source projects at https://github.com/hso-esk/tls-diff-testing.

Acknowledgments

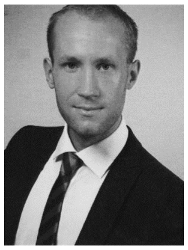
We are very grateful for the valuable feedback we received from the anonymous reviewers of our manuscript as well as from our colleagues Manuel Schappacher, Artem Yushev, and Oliver Kehret. We furthermore would like to thank Prof. Dr. Dirk Timmermann from Rostock University for his scientific support of our work.

References

[1] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2," RFC 5246, Aug. 2008.
[2] R. Oppliger, *SSL and TLS: Theory and Practice*. Artech House Publishers, 2009.
[3] L. C. Paulson, "Inductive Analysis of the Internet Protocol TLS," *ACM Transactions on Information and System Security*, vol. 2, no. 3, pp. 332–351, Aug. 1999.
[4] G. Díaz, F. Cuartero, V. Valero, and F. Pelayo, "Automatic Verification of the TLS Handshake Protocol," in *Proc. ACM Symposium on Applied Computing (SAC '04)*, 2004, pp. 789–794.
[5] K. Ogata and K. Futatsugi, "Equational Approach to Formal Analysis of TLS," in *Proc. 25th IEEE International Conference on Distributed Computing Systems (ICDCS '05)*, Jun. 2005, pp. 795–804.
[6] S. Gajek, M. Manulis, O. Pereira, A.-R. Sadeghi, and J. Schwenk, "Universally Composable Security Analysis of TLS," in *Proc. 2nd International Conference on Provable Security (ProvSec 2008)*, Oct. 2008, pp. 313–327.
[7] H. Krawczyk, K. G. Paterson, and H. Wee, "On the Security of the TLS Protocol: A Systematic Analysis," in *Proc. 33rd Annual Cryptology Conference (CRYPTO 2013)*, Aug. 2013.
[8] C. Meyer and J. Schwenk, "Lessons Learned From Previous SSL/TLS Attacks - A Brief Chronology Of Attacks And Weaknesses," Cryptology ePrint Archive, Report 2013/049, Jan. 2013.
[9] Y. Sheffer, R. Holz, and P. Saint-Andre, "Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)," RFC 7457, Feb. 2015.
[10] ——, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," RFC 7525, May 2015.
[11] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proc. ACM Conference on Computer and Communications Security (CCS '12)*, Oct. 2012, pp. 38–49.
[12] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell, "Not-Quite-So-Broken TLS: Lessons in Re-Engineering a Security Protocol Specification and Implementation," in *Proc. 24th USENIX Security Symposium (USENIX Security 15)*, Aug. 2015, pp. 223–238.
[13] "OpenSSL 'Heartbleed' vulnerability," CVE-2014-0160, 2013.
[14] "OpenSSL 'CCS injection' vulnerability," CVE-2014-0224, 2013.
[15] "OpenSSL: TLS/SSL and crypto library," www.openssl.org.
[16] "GnuTLS Certificate verification vulnerability," CVE-2014-0092, 2014.
[17] "GnuTLS: GNU Transport Layer Security Library," www.gnutls.org.
[18] S. Bratus, T. Darley, M. Locasto, M. L. Patterson, R. B. Shapiro, and A. Shubina, "Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier," *IEEE Security and Privacy*, vol. 12, no. 1, pp. 83–87, 2014.
[19] D. A. Wheeler, "How to Prevent the next Heartbleed," Apr. 2014.
[20] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Y. Strub, "Implementing TLS with Verified Cryptographic Security," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 445–459.
[21] "miTLS: A Verified Reference Implementation of TLS," mitls.org.
[22] 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 *Proc. 2014 IEEE Symposium on Security and Privacy (SP '14)*, May 2014, pp. 114–129.
[23] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.
[24] Y. Chen and Z. Su, "Guided Differential Testing of Certificate Validation in SSL/TLS Implementations," in *Proc. 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, Aug. 2015, pp. 793–804.
[25] T. Petsios, A. Tang, S. J. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient Domain-independent Differential Testing," in *Proceedings of the 38th IEEE Symposium on Security & Privacy*, San Jose, CA, May 2017.
[26] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "HVLearn: Automated Black-box Analysis of Hostname Verification in SSL/TLS Implementations," 2017.
[27] "American Fuzzy Lop (AFL)," http://lcamtuf.coredump.cx/afl.
[28] J. Somorovsky, "Systematic Fuzzing and Testing of TLS Libraries," in *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, Oct. 2016, pp. 1492–1504.
[29] "`tls-diff-testing`: Differential Handshake Fuzz-Testing of TLS Implementations," https://github.com/phantax/tls-diff-testing.
[30] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, Jan. 1999.

19. A note to the reviewers of our manuscript: our software tools will be made publicly available if and as soon as our manuscript is accepted for publication. You can find the source code of our tools attached to our submission.

[31] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” draft-ietf-tls-tls13-20, Apr. 2017, Work in Progress.
[32] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, *Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols*, 2015.
[33] G. Shu and D. Lee, “Testing security properties of protocol implementations - a machine learning based approach,” in *Distributed Computing Systems, 2007. ICDCS '07. 27th International Conference on*, June 2007, pp. 25–25.
[34] G. Shu, Y. Hsu, and D. Lee, “Detecting communication protocol security flaws by formal fuzz testing and machine learning,” in *Formal Techniques for Networked and Distributed Systems FORTE 2008*, ser. Lecture Notes in Computer Science, K. Suzuki, T. Higashino, K. Yasumoto, and K. El-Fakih, Eds. Springer Berlin Heidelberg, 2008, vol. 5048, pp. 299–304.
[35] Y. Hsu, G. Shu, and D. Lee, “A model-based approach to security flaw detection of network protocol implementations,” in *Network Protocols, 2008. ICNP 2008. IEEE International Conference on*, Oct 2008, pp. 114–123.
[36] W. Tang, A.-F. Sui, and W. Schmid, “A model guided security vulnerability discovery approach for network protocol implementation,” in *2011 IEEE 13th International Conference on Communication Technology*, Sept 2011, pp. 675–680.
[37] J. de Ruiter and E. Poll, “Protocol State Fuzzing of TLS Implementations,” in *Proc. 24th USENIX Security Symposium (USENIX Security 15)*, Aug. 2015, pp. 193–206.
[38] 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 *Proc. 2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 535–552.
[39] H. Kario, “TLSFuzzer: TLS Test Suite and Fuzzer,” www.github.com/tomato42/tlsfuzzer.
[40] “TLS-Attacker: A Java-based Framework for Analyzing TLS Libraries,” https://github.com/RUB-NDS/TLS-Attacker.
[41] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias, “SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning,” in *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, Oct 2016.
[42] “libFuzzer: A Library for Coverage-Guided Fuzz Testing,” http://llvm.org/docs/LibFuzzer.html.
[43] A. Walz and A. Sikora, “eTPL: An Enhanced Version of the TLS Presentation Language Suitable for Automated Parser Generation,” in *9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, 2017.
[44] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, “Bug Characteristics in Open Source Software,” *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.
[45] “BoringSSL: A Fork of OpenSSL that is Designed to Meet Google's Needs,” boringssl.googlesource.com/boringssl.
[46] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS),” RFC 4492, May 2006.
[47] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig, “Transport Layer Security (TLS) Session Resumption without Server-Side State,” RFC 5077, Jan. 2008.
[48] R. Seggelmann, M. Tuexen, and M. Williams, “Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension,” RFC 6520, Feb. 2012.
[49] J. Postel, “DoD standard Transmission Control Protocol,” RFC 761, Jan. 1980.
[50] E. Allman, “The Robustness Principle Reconsidered,” *Commun. ACM*, vol. 54, no. 8, pp. 40–45, Aug. 2011.
[51] M. Thomson, “The Harmful Consequences of Postel's Maxim,” draft-thomson-postel-was-wrong-00, Mar. 2015, Work in Progress.
[52] K. Bhargavan and G. Leurent, “Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH,” in *Proc. Network and Distributed System Security Symposium (NDSS 2016)*, Feb. 2016.
[53] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, “Exploit Programming: From Buffer Overflows to “Weird Machines” and Theory of Computation,” *;login:*, vol. 36, no. 6, 2011.
[54] A. Freier, P. Karlton, and P. Kocher, “The Secure Sockets Layer (SSL) Protocol Version 3.0,” RFC 6101, Aug. 2011.
[55] A. Zeller and R. Hildebrandt, “Simplifying and Isolating Failure-Inducing Input,” *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, Feb. 2002.
[56] C. Meyer, “20 Years of SSL/TLS Research: An Analysis of the Internets Security Foundation,” Ph.D. dissertation, Univ. of Bochum, Feb. 2014.



Andreas Walz holds a diploma in physics from the University of Freiburg. From his studies and from working as a research assistant in experimental particle physics he has many years of experience in the development of embedded hardware systems as well as efficient software. Currently, he is pursuing his Ph.D. in the field of security in embedded systems with special interest in the TLS protocol family.

Axel Sikora holds a diploma of electrical engineering and a diploma of business administration, both from Aachen Technical University. He has done a Ph.D. in electrical engineering at the Fraunhofer Institute of Microelectronics Circuits and Systems, Duisburg, with a thesis on SOI-technologies. After positions in the telecommunications and semiconductor industry, he became a professor at the Baden-Wuerttemberg Cooperative State University Loerrach in 1999. In 2011, he joined Offenburg University of Applied Sciences, where he leads the Institute of Reliable Embedded Systems and Communication Electronics. Since 2016, he is also deputy member of the board at Hahn-Schickard Association of Applied Research, where he heads the “Software Solutions” division. His major interest is in the field of efficient, energy-aware, safe, and secure algorithms and protocols for wired and wireless embedded communication. In 2002, he founded the Steinbeis Transfer Center Embedded Design and Networking for professional protocol and platform developments, which was successfully spun off as STACKFORCE GmbH in 2014. Dr. Sikora is the author, co-author, editor, and co-editor of several textbooks and numerous papers in the field of embedded design and wireless and wired networking, and head and member of manifold steering and program committees of international scientific conferences.