# eGMT-Fuzz: Format-Aware Deep Fuzzing of Cryptographic Protocols

Angel Lomeli
Aalto University
Espoo, Finland
angel.lomeli5@proton.me

Arto Niemi
Huawei Technologies Oy
Helsinki, Finland
arto.niemi@huawei.com

***Abstract*—Fuzzing has established itself as an everyday tool in the toolbox of the security-minded software developer. Fuzzers have proven especially effective in discovering vulnerabilities that are rarely triggered during regular program execution. Interactive cryptographic protocols, however, are challenging to fuzz. Messages in such protocols must pass cryptographic validation such as integrity and freshness checks, before execution can reach deeper portions of the protocol implementation code.**

**In this paper, we present a black box mutation-based fuzzer for deep fuzzing of interactive cryptographic protocols. To create messages that mostly conform to the protocol syntax but are syntactically or semantically unexpected, we use syntax tree mutation. Our architecture includes a pluggable component that allows mutated inputs to pass protocol-specific cryptographic checks. We evaluate the efficacy of our fuzzer on an embedded Transport Layer Security (TLS) implementation, where we deeply fuzz both TLS handshake messages and X.509 public-key certificates, discovering several hard-to-reach vulnerabilities.**

## I. Introduction

*Fuzzing* is a form of software testing where syntactically or semantically incompliant input messages are generated randomly, either by mutating valid seed messages or by applying grammar-based production rules. The messages are then fed to the program under test, which is monitored for unexpected behaviour such as crashing or buffer overflows. Fuzzing is mainly used in security testing, where tools such as American Fuzzy Lop (AFL) have proven to be extremely efficient in discovering vulnerabilities in programs that parse input files, while network-oriented fuzzers such as AFLNet allow implementations of interactive protocols to be fuzzed more efficiently.

However, fuzzing implementations of interactive *cryptographic* protocols, such as Transport Layer Security (TLS) [1], present a special challenge that is not sufficiently addressed by existing methods and tools. Such protocols contain lengthy message flows (handshakes) where messages are required to pass cryptographic checks such as signature verifications and MACs before being processed further. Traditional fuzzers often perform poorly in such situations—they tend to generate inputs that cause the protocol run to fail too early. An integrity-protected octet has only one legal value, so a random mutation in the octet results in the message being discarded – often even before parsing – and in the termination of the handshake. Format-aware fuzzers, based on e.g. syntax tree mutation, can be more efficient in finding parsing errors, but they also suffer from the failure to pass cryptographic checks. Therefore, a format-aware fuzzer with a *protocol termination capability*, that can apply mutations *before* encryption and do cryptographic operations, such as key derivation and re-signing to pass cryptographic checks, seems to be the best approach for *deep fuzzing* of such protocols.

In this paper, we develop such a mutation-based, format-aware and crypto-capable fuzzer for interactive cryptographic protocols. Our starting point is the Generic Message Tree (GMT) method of Walz and Sikora [2], which works by dissecting protocol messages into a parse tree structure, subjecting the nodes to fuzz operations, repairing the mutated nodes' ancestors to restore their format compliance, serializing the tree back to the protocol encoding, and transmitting the result as input to the program under test.

Our work improves the baseline of the Walz-Sikora fuzzer in two directions. First, we add new GMT fuzz operators and optimize existing ones. Second, we add protocol termination capability in order to reach latter portions of the TLS handshake. This allows us to fuzz the whole protocol and not just the initial `ClientHello` message. We validate our design with a fuzzing campaign against a development version of a custom, presently closed-source, TLS library, from which we were able to find multiple issues that would have been difficult or impossible to discover using conventional fuzzers with format-breaking random mutations. Our results are in line with earlier work [3], which also noted the benefits of format-aware fuzzing with the GMT in the security testing of TLS implementations.

The rest of the paper is organized as follows. The background section introduces fuzzing, the TLS 1.3 protocol and the format of messages transmitted in a typical TLS handshake. Next, we introduce the enhanced GMT (eGMT) data structure and novel fuzz operators that we apply to its nodes. We describe our fuzzing architecture and the protocol termination ("man-in-the-middle") component and show how the fuzzer can be applied to TLS handshakes and X.509 certificates. After this, we present our findings. Finally, we provide conclusions and ideas for further work.

## II. Background

### A. Fuzzing

The term "fuzzer" was coined by Miller et al. in 1990 [4]. In their report, the authors discovered they could crash 25-33% of the Unix utility programs they tested [5] by using simple black-box testing with randomly generated inputs. The method of randomly generated test inputs is much older, however; the article of Duran et al. from 1981 is an important example of such early work [6]. The introduction of American Fuzzy Lop (AFL) in 2015 provided a significant stimulus to research and industry efforts [7, p. 1201]. Today, fuzz testing is routinely used in software engineering. For example, the Microsoft Secure Development Lifecycle mandates that untrusted interfaces of all products must be subjected to fuzzing [8], and the Android ecosystem includes an extensive fuzzing infrastructure [9] for code submissions. The core ideas behind fuzz testing are remarkably simple, yet extremely effective in practice. (In a later paper, Miller revealed that the authors of [4] initially had trouble finding a publisher, partly because their work was regarded as "too simplistic".) In the following, we briefly survey the taxonomy of fuzzers and the challenges one faces when attempting to apply them to cryptographic protocols.

**Color**. Fuzzers are traditionally assigned a color [8] according to the amount of knowledge they require of the *program under test* (PUT). *Blackbox* fuzzers such as Miller's original *fuzz* tool only look at the input-output behaviour of the PUT; they do not require binary instrumentation or access to the source code. *Whitebox* fuzzers [10] use dynamic symbolic execution to discover branch conditions and then apply constraint solving to find inputs that allow reaching untested branches. This allows whitebox fuzzers, in theory, to systematically test all executions paths of the PUT, although they are often very slow in practice [7]. *Greybox* fuzzers such as AFL constitute the middle ground between black and whitebox fuzzers. They typically rely on lightweight code instrumentation that produces code coverage statistics at runtime, and use coverage to guide input generation [7]. In contrast to whitebox fuzzers, greybox fuzzers cannot systematically cover all execution paths, but offer much better performance.

**Input generation**. A fuzzer can also be classified according to its input production method. *Generation-based* fuzzers create the inputs from scratch, using a formal grammar or a specification, such as the file format or protocol syntax. A drawback of the generational fuzzers is that they either require bespoke message generation code for target format or a formal grammar, which is rarely available, especially for cryptographic protocols. In contrast, *mutation-based* fuzzers create inputs by iteratively applying random changes such as bit flips to messages, starting from a valid seed message.

**Format-awareness**. Traditional mutation-based greybox fuzzers, such as AFL, excel at fuzzing programs that process inputs with little syntactic structure, for example images. When applied to highly structured inputs, such as protocol

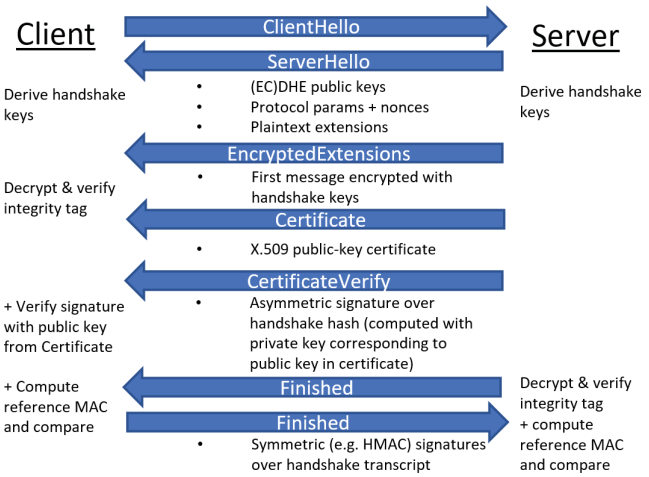


Fig. 1. A typical TLS 1.3 handshake with main message contents and cryptographic integrity checks. The failure of any of the checks results in immediate abortion of the handshake according to protocol specification. This makes it difficult for fuzzers to reach latter parts of the handshake.

messages, these fuzzers produce inputs that are rejected in the parsing stage without reaching later stages such as semantic checking and application processing. To address this, several *format-aware* or *smart* [11] fuzzers have been proposed. These are able to direct mutations to specific syntactic elements, minimizing or avoiding syntax violations. One way to achieve format-awareness is to use dictionaries from which valid byte sequences can be chosen and injected at random locations in the mutated data [11]. However, dictionary construction is often complex and dictionary-based replacement fails to account for context, such as dependencies between protocol messages. Another approach is to parse the input into a syntax tree and then apply mutations to the tree nodes [12]. Superion [13] mutates the tree by replacing a random subtree with a subtree from a tree that was constructed from earlier data. This suffers from the same problem as dictionary-based replacement. The Walz-Sikora fuzzer [2], described in Section II-D, goes beyond subtree replacement and applies type-specific mutations to the syntax tree nodes.

**Challenges**. While standard file processing tools and protocols work in essentially three stages (syntax parsing, semantic checking, application execution), a cryptographic protocol such as TLS [1] (described in more detail below) adds cryptographic operations such as decryption and integrity validation. These are usually performed before syntax parsing. For example, TLS uses authenticated encryption for all protocol messages after the initial ClientHello and ServerHello messages. Messages must be decrypted and checked against the integrity checksum before parsing. A further challenge is that the implementation under test should be monitored not only for crashes, but also for protocol-incompliant behaviour, such as failure to turn on encryption or to abort the handshake upon reception of an unexpected message.

### B. TLS handshake

Transport Layer Security (TLS) is a secure channel protocol [14, pp. 88-91] consisting of an authenticated key exchange and subsequent transmission of encrypted and integrity-protected data. TLS is used to protect a vast majority of Internet, especially HTTP, traffic. Here, we focus on the latest TLS 1.3 version [1]. As illustrated in Fig. 1, a typical TLS handshake, where only the server is authenticated, consists of seven messages. The first two, ClientHello and ServerHello, are used to negotiate an (EC)DHE shared secret and to agree upon protocol parameters. The handshake is integrity-protected in three ways. First, all messages after the ClientHello and ServerHello are encrypted and integrity-protected with an authenticated encryption (AE) cipher such as AES-GCM, using handshake traffic protection keys derived from the shared secret and a hash of the first two messages. Second, the CertificateVerify message includes an asymmetric (e.g. ECDSA) signature over a hash of the previous handshake messages. Third, the Finished message provides a final integrity protection by including a symmetric (e.g. HMAC) signature over the preceding handshake messages. Thus, it is clear that simple black-box mutation of intercepted messages will result in a failure in one these three types of integrity checks.

### C. Format and encoding of TLS messages

The TLS protocol specifies the format and encoding of its messages using the domain-specific TLS presentation language (TPL) [1, Section 3]. With TPL, messages are defined using a syntax that closely resembles how structs are defined in the C programming language. TPL also specifies a simple encoding, with variable-size vectors prefixed with a Big Endian encoding of their length. The following is a simple, artificial example of a TPL-defined structure:

```
struct {
  uint16_t id;
  opaque data<1..2^16-1>;
} InnerMessage

struct {
  uint8_t version;
  uint16_t id;
  InnerMessage messages<1..2^16-1>;
} TPLExampleMessage;
```

Here, `data` and `messages` are variable-length vectors, defined to contain 1 to $2^{16} - 1$ bytes. The actual length is appended to the encoding as a prefix.

Assuming that `TPLExampleMessage` has `version:1`, `id:51966` (0xcafe in hexadecimal), and that the `messages` vector contains a single `InnerMessage` with `id:2` and `data:0xfacafe`, the whole `TPLExampleMessage` would be encoded as `0x01cafe000700020003facafe`. Here, $2 = \log_2 2^{16} - 1$ Big Endian bytes (`0003`) are needed to encode the length of the data vector in the inner message, since it is defined to hold at most $2^{16}-1$ octets. Furthermore, the `messages` vector is prefixed with the byte-length of its contents in Big Endian encoding. This is `0007` since the single InnerMessage is encoded with the seven bytes `00020003facafe`.

For the X.509 public-key certificates that are transmitted in the TLS' Certificate handshake messages, as well as for the ECDSA signatures transmitted in the CertificateVerify message as a proof-of-possession of the private authentication key, another format, called Abstract Syntax Notation One (ASN.1) [15], is typically used. ASN.1 actually has multiple possible encodings, including XML- and JSON-based ones (XER and JER, respectively), but in the security domain the strict, binary Distinguished Encoding Rules (DER) are typically used, since it minimizes the amount of encoder options. This is important for signed data, the encoding of which should be unique. The following is a real-world example: a typical encoding of an ECDSA signature in TLS, with rarely used optional fields removed [16, p. 114].

```
ECDSA-Signature ::= SEQUENCE {
  r INTEGER,
  s INTEGER,
}
```

An example DER-encoded ECDSA-Signature object is shown in Fig. 2. The DER-encoding of ECDSA-Signature starts with a tag (0x30) identifying the outer type (SEQUENCE). The tag value also includes a 1-bit indicating that what follows is not a "leaf" item, but a constructed object which itself consists of further items (two integers). Next comes the length of the ECDSA-Signature value part. The two integers are also both encoding using tag, length and value – first the INTEGER tag 0x02, then a length octet and finally the value (a coordinate on the elliptic curve).

For our purposes in the context of format format-aware fuzzing, we make two important observations. First, ASN.1/DER is a tag-length-value (TLV) encoding, where the value part (V) can consist of further TLVs. Second, TPL objects can similarly be nested: even though their encoding does not include an explicit type (T), variable-length vectors are encoded as an "LV" with a length indicator (L) prepended to the value (V). Both formats are amenable to tree-like representations.

### D. Generic Message Trees

In 2020, Walz and Sikora [2] proposed a black-box, mutation-based, format-aware fuzzer (later referred to as *tls-diff* in [17]). Combined with differential testing, the fuzzer allowed the authors to find vulnerabilities in several TLS implementations. The format-aware component in the Walz-Sikora fuzzer is a dynamic data structure called *Generic Message Tree* (GMT) that supports a set of fuzz operators on the tree nodes. Walz and Sikora proposed eight such operators, using them to generate mostly-valid TLS 1.2 ClientHello messages by first parsing a valid seed message into a GMT and randomly applying fuzz operations to the nodes, and finally

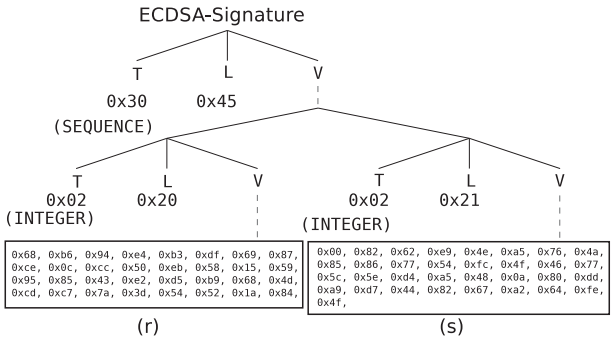


Fig. 2. GMT of a DER-encoded ECDSA-Signature

re-encoding the message. To detect erroneous behaviour such as crashing or protocol violations, the authors compared the responses of various TLS implementations to the fuzzed input. Their GMT-based approach achieved better code coverage and found more response discrepancies than AFL, TLS-Attacker [18] and NEZHA [19]. Walz and Sikora only fuzzed the initial ClientHello message in the TLS handshake, but saw their work as the first step towards full interactive fuzzing of the protocol. (In their conclusions, Walz and Sikora stated: "we see our approach as presented herein only as the first step towards *fully interactive* differential testing of black-box TLS protocol implementations" [2, p. 289].) Later, Pan et al. [17] optimized the differential testing mechanism in the Walz-Sikora fuzzer, without changing the GMT operators or adding new ones.

**Definition**. A GMT is an ordered rooted tree with typed nodes. Leaves represent atomic data such as integers or raw unstructured octets. Internal nodes represent composite types such as vectors. A protocol message can be translated into its GMT representation via a protocol-specific *dissection* function $D$. Converting a GMT to the on-the-wire representation can be done by invoking a *serialization* function $S$ that traverses the tree in depth-first order and encodes each leaf node according to its type.

**Example**. Fig. 2 shows a GMT dissected from an ECDSA signature, encoded with Distinguished Encoding Rules (DER) DER is a Tag-Length-Value (TLV) encoding [15]: the tag T encodes the type of the value V, and L encodes the number of octets in V. The value can also consist of another TLV. The ECDSA signature is a sequence of two integers $r$ and $s$. Each T and L, as well as $r$ and $s$ are represented by leaves, while the value of the sequence TLV is an internal node. Note while the GMT concept is generic, the code by Walz and Sikora only supported TLS ClientHello messages. Adding support for ASN.1 and DER was done during our present work.

**Benefits**. The usefulness of the GMT for fuzzing comes from the fact that each node can be mutated with or without changing the node's type or violating structural constraints. For example, an integer node can be mutated into another integer or into a string, vector elements can be removed or duplicated, a subtree rooted at an internal node may be truncated, etc. Walz and Sikora proposed eight such *GMT operators*, summarized in Table I. Some of the operators can be applied on nodes of a particular type, accomplished using *filters* that return applicable nodes.

TABLE I. THE ORIGINAL GMT OPERATORS FROM [2, SECTION 5.1.1]

| Operator | Applies to | Function |
|---|---|---|
| DelOp | Vectors | Deletes the node |
| VoidOp | Vectors | Deletes the node, adds a placeholder |
| DupOp | Dynamic length elements | Adds a copy of the node as its sibling |
| TruncFuzzOp | Dynamic length elements | Chooses a random smaller length, truncating node bytes up to this length |
| IntFuzzOp | Leaves | Replaces node with an integer N. Proximity mode: representation of N resembles original. Full range mode: N is chosen randomly from the original integer range. The mode is chosen at random |
| DataFuzzOp | Leaves | Replaces the node with a random byte array of the same length as the original |
| AppFuzzOp | Dynamic length elements | Generates a new byte array of up to four bytes and appends it as a sibling |
| GenFuzzOp | CipherSuites vector, Extensions vector, or any Extension element | Replaces a node or subtree with semi-random data that follows the same structure as the original |

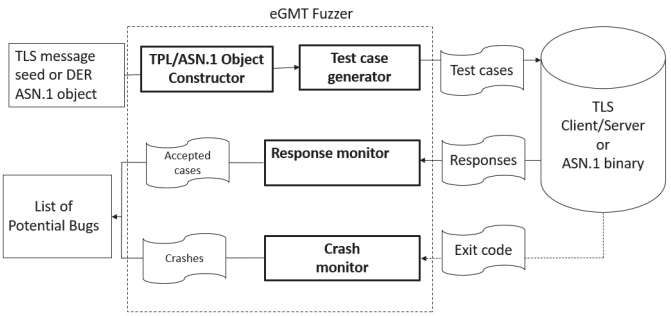


Fig. 3. eGMT-Fuzzer components

**Operators**. The Walz-Sikora fuzzer included eight fuzz operators for GMT nodes and subtrees, shown in Table I. On every iteration, the fuzzer first selects an operator and filters out nodes whose type is compliant with the operator. From the remaining nodes, the fuzzer selects a random node and applies an operator to the node. Finally, the fuzzer chooses whether to repair the length fields from the target node up to the root node. At the end of each iteration, the fuzzer decides at random whether to execute a new iteration or not with a probability of 1/2.

## III. ARCHITECTURE

The architecture of our fuzzer, which we call *eGMT-Fuzz*, is shown in Fig. 3. Like the Walz-Sikora fuzzer, it is based on GMTs, but, as described in Section IV, we add enhancements such as new operators and cryptographic capabilities; we call the result *enhanced Generic Message Tree* or eGMT.

First, a valid handshake message that corresponds to the message to fuzz (except for the `CertificateVerify` and `Finished` messages, which are generated by the fuzzer automatically) or a DER-encoded ASN.1 structure is used as seed. This seed is sent to the correct dissection function (*TPL Object Constructor* or *ASN.1 Object Constructor*) depending on the object structure to be fuzzed. When both constructors need to be used, such as for handshake messages that contain ASN.1 structures, the seed is first parsed by the TPL Constructor and then, the specific fields containing the ASN.1 structure in the TPL object are parsed by the ASN.1 constructor, replacing the bytes they originally pointed to with a child ASN.1 eGMT. The constructor is also the component responsible for telling the fuzzer how to send the test cases to the PUT and how to interpret the responses.

The parsed eGMT is then sent to the *Test case generator*, where it is modified randomly as shown in Algorithm 1. The fuzzer then executes the PUT (a TLS client or server when the handshake is being fuzzed, or a binary that takes ASN.1 structures as input for parsing when the fuzz target is the ASN.1 parser) as a subprocess and either sends the final test case to the subprocess over the local network or instructs the PUT to open the test case from a file. The *Response monitor* keeps track of the responses sent by the PUT on the network when applicable, while the *Crash monitor* checks the exit code of the PUT after finishing the execution. The *Response monitor* performs different checks based on the message being fuzzed to decide on whether a test case was accepted or rejected by the PUT. The *Crash monitor* saves any test cases that triggered an exit code different than 0 (successful execution with no errors) or 1 (failed execution caused by an expected error). Both of these monitors then generate a list of the test cases that caused interesting behaviors and need manual analysis.

## IV. Enhanced Generic Message Trees

The original GMT fuzzer by Walz and Sikora is able to fuzz only the `ClientHello` message in a TLS handshake. Furthermore, it works only for TLS version 1.2 and below, excluding the latest version TLS 1.3, at the time of writing. Using their work as a basis, we developed an extended version of GMTs called Enhanced Generic Message Trees (eGMTs), which we used to further fuzz the TLS protocol, including more handshake messages. This was achieved by adding key derivation capabilities in the fuzzer, allowing us to fuzz messages before encryption, decrypt received responses, and generate valid integrity codes. Currently, eGMT-Fuzz supports TLS 1.3 and X.509 public-key certificates. It is able to fuzz the following TLS 1.3 handshake messages: `ClientHello`, `ServerHello`, `EncryptedExtensions`, `Certificate`, `CertificateVerify`, and `Finished`.

For the parsing of handshake messages, we created a collection of objects based on the TLS Presentation Language (TPL) composed of Python dictionaries and lists that dictate the tree-to-bytes correspondence. These objects tell the fuzzer how many bytes to read for each field and how to interpret the data, taking into account DER-encoded length fields and children subtypes. As in the original GMT implementation, a parsed message becomes a tree-like structure with nodes that contain either additional children nodes or message bytes.

### A. New and improved operators

The original GMT operators described by Walz and Sikora were enough to generate several variations of the ingested corpus. However, we noticed that they could be changed to improve their mutation capability and to generate more interesting test cases. In the Walz-Sikora fuzzer, the filter function governs which operators can be applied to which types of node. For example, the filter allows the duplication operation to apply only to subtrees, but not to leaf nodes. We found the filtering to be excessively restrictive; for example, duplicating leaf nodes can also be useful. Consequently, a major change we did was to make all but the Integer Fuzz Operator applicable to any type of node or subtree. This rendered most of the filters unnecessary, and ensuring that potentially useful modifications are not prevented. The only filters we kept were the Leaf Node Filter and the Vector Element Filter.

In addition, we noticed that the behavior of some of the operators could be improved. The Integer Fuzz Operator in ”full range mode” does exactly the same actions as the Content Fuzz Operator, which is set to maintain the original length. The Voiding and Deleting operators behave in the same way with the exception of the place marker deletion. With these remarks in mind, we implemented an ”enhanced” version for each operator, as shown in Table II.

We also introduced three new operators, shown in Table III. The ZeroOp operator, which zeroes bytes in the target node, was based on the observation that zero often has a special meaning: length indicator with value 0 indicates (seemingly) missing data, and in public-key cryptography 0 often constitutes a special value. The rotation operator, loosely inspired by the self-balancing AVL trees [20], changes the order of two sibling nodes in the eGMT. In TLS, the order in which the handshake messages must be sent is fixed, and transmitting some messages (such as Finished) too early may lead to cryptographic operations performed at the wrong time. Finally, the simple BitFlip operator, which turned out to be one of the most effective ones in our tests, simply flips up to five randomly chosen bits in a node.

Given that these last three operators do not affect length, the length repair decision is skipped for them (as well as for the Integer Fuzz Operator).

### B. Fuzzing X.509 certificates

The GMT fuzzing approach was originally intended to fuzz TLS implementations. Our eGMT-Fuzz also focuses on TLS, but we developed a module that uses the same principle to mutate DER-encoded ASN.1 structures such as X.509 certificates, showing that it can be easily applied to other similar hierarchical structures. Our module parses the ASN.1 objects in a ”dumb” way following the DER syntax without looking at the context of each particular case. It simply looks

TABLE II. ENHANCED GMT OPERATORS IN EGMT-FUZZ

| Operator | Applicable nodes | Description |
|---|---|---|
| eVoidOp | Any | Deletes a node while keeping a place marker |
| eDupOp | Any | Creates a copy of a node and adds it as a sibling of the original |
| eTruncFuzzOp | Any | Chooses a new random length lower than the original and truncates the bytes in the node to match it. If the node has children, they are also removed or truncated iteratively |
| IntFuzzOp | Leaf nodes | In "proximity mode", replaces the node with a random integer that is close to the integer representation of the original value. In "full range" mode, the new integer is chosen up to the maximum value allowed for the byte length. The mode to use is also chosen at random. |
| eDataFuzzOp | Any | Replaces the node with a random byte array of the same length as the original or a new length of at most the double of the original |
| eAppFuzzOp | Any | Generates a new byte array of up to four bytes or a new element from a list of valid data for the specific target node and adds it as a sibling |
| eGenFuzzOp | Any | Replaces a node or subtree with semi-random data that follows the same structure as the original, if applicable. If not, behaves just like the eContent-FuzzOp |

TABLE III. ADDITIONAL OPERATORS IN EGMT-FUZZ.

| Operator | Applicable nodes | Description |
|---|---|---|
| ZeroOp | Leaf nodes | Replaces all bytes in the target node with a 0x00 byte array |
| BitFlipOp | Leaf nodes | Flips up to five random bits in a node |
| RotOp | Vector elements | Chooses a new random sibling of the same type of the target node and switches their positions |

at the tag byte and checks whether the constructed/primitive bit is set (indicating it has children, i.e., the Value field is another TLV) or not (indicating it simply contains bytes), and creates a tree-like structure made out of TLV subtrees. Every tag and length field is a leaf in the tree, while the value field can be either a leaf of bytes or another TLV object as a child. After parsing, the ASN.1 eGMTs were able to work with the same operators described earlier, with a couple exceptions for which a custom operator was created:

- **ASN.1 Appending Fuzz Operator.** Creates a new subtree to append as a sibling of the operand. In ASN.1 eGMTs, however, the new subtree must also be a TLV object. The tag is randomly chosen to be either one of the most common ASN.1 types used in TLS or a random byte. The length is chosen at random as well and it indicates how many bytes to generate for the value field.
- **ASN.1 Synthesizing Fuzz Operator.** Since our ASN.1 parser looks only at the DER encoding and does not care about object context, there are no lists of valid constant values to generate semi-valid data. This operator also generates a new TLV object in the same way as the previous operator, but uses it to replace the operand.

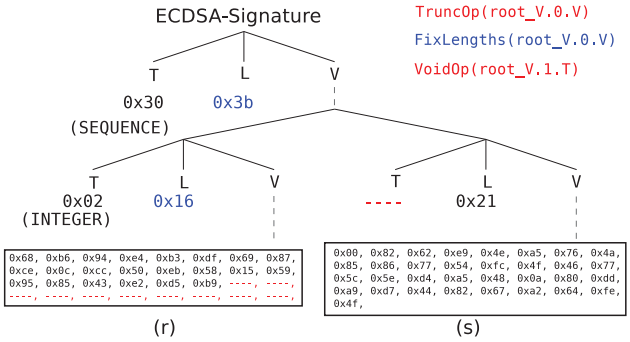


Fig. 4. Example application of random fuzz operators. First, the value of the `r` TLV is truncated, and the lengths of the affected subtree are repaired. Finally, the tag of `s` is removed without fixing lengths.

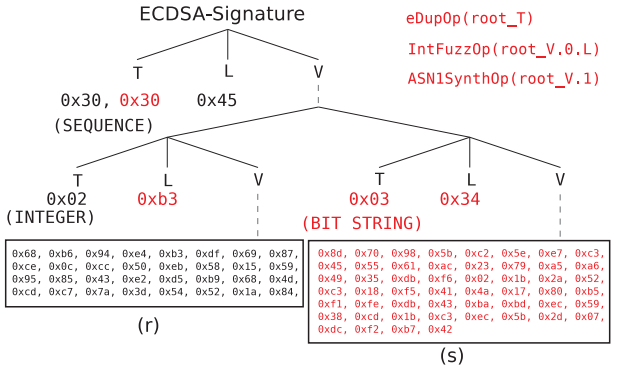


Fig. 5. Here, the duplication operation is applied to the SEQUENCE tag of the top-level TLV. Next, the integer fuzz operator is applied to the length octet of the `r` TLV. Finally, the ASN.1 synthesizing operator is applied to `s`, fuzzing the tag, length and the value.

Furthermore, since the length bytes in TLV objects need to comply with additional rules, the repairing algorithm was modified to detect when the underlying object was a TLV or TPL object. Once a length is repaired, the algorithm will encode it according to the detected object before changing the corresponding fields.

The ASN.1 module was used to generate and mutate eGMTs using as seeds: ECDSA signatures, SubjectPublicKeyInfo objects, and X.509 certificates. The fuzzer would write each mutated test case to a file which was read by a binary that called functions from the a TLS library for, e.g., signature verification or generation of ECDH shared secrets. With this approach, the fuzzer found a set of bugs in the parsing functions used by the library. The bugs allowed, for example, to send an arbitrary amount of garbage bytes at the end of an object or to crash the application altogether when the length field was invalid.

### C. Deep fuzzing of the TLS 1.3 handshake

eGMT-Fuzz includes a set of modules for fuzzing each of the handshake messages. The message flow is different for each and the fuzzer needs to perform different checks depending on the fuzzed message to decide whether a bug exists or not. First, the target message is selected. This message is the only one that will be mutated for a particular session. A seed message is fed into the fuzzer, which generates a new test case by randomly applying mutations on the seed. Then, the fuzzer performs a handshake by simulating the actions of a client or server, replacing the target message with its mutated version. Finally, depending on the responses (or lack of them) it gets, it categorizes the generated test case as accepted, rejected, or crash. The fuzzing process works on the premise that the applied mutations will generate invalid payloads, so test cases considered rejected are the only ones that are not saved to disk for later inspection, as this is considered to be normal behavior. Fig. 6 shows the pseudo-code for the generation of new test cases based on seed messages, while the length repairing function is shown in Fig. 7.

Due to the complexity of the TLS protocol, looking for crashes in the binaries is not enough. It is also important to see if the handshake continues even after an invalid message was sent or if the execution flow is stopped at a later state. If the fuzzer determines that a fuzzed message was accepted, it saves the message to a file for later examination and the hash of the message in a list of hashes that is checked to avoid duplicating test cases. The fuzzer also has the capability to read the exit code of the client or server binaries. If it finds an abnormal exit code, the generated test case and its hash are saved in another location meant for potential crashes.

The mutations are generally applied to valid handshake messages used as seeds. However, for both the `CertificateVerify` and `Finished` messages this is not the case. These messages contain signatures or HMACs that are generated using information specific to each handshake, e.g., the transcript hash of all exchanged messages. For this reason, the corresponding fuzzing modules must first perform the handshake up to the point where the target message will be sent, generate a valid message, and then perform the mutations on the generated message.

It is often the case that, given the random nature of the algorithm, a specific mutation is reversed, resulting in a generated test case that is identical to the original seed. This happens when, e.g., a length field is fuzzed but the repairing algorithm fixes it back to the original, or a recently appended node is deleted by the eVoiding Operator. This results in a false positive test case that is considered accepted but does not indicate the potential existence of a bug. For most messages, this false positive is written just once, since the hash of the generated case is checked before writing. For the `CertificateVerify` and `Finished` messages, however, it becomes more problematic given that the messages are different for each handshake, and thus the hashes are also different. The amount of false positives that are written to files as findings for these messages is significantly larger, making manual inspection necessary.

The eGMT TPL and ASN.1 modules were combined as well for an additional set of tests with messages that included ASN.1 structures. This was particularly helpful for the `Certificate` and `CertificateVerify` messages, which can contain ECDSA signatures and X.509 certificates. For these cases, the parser would first create a tree with the TPL structure and then create subtrees for any ASN.1 structures as children. After this, the fuzzer is able to perform mutations on nodes from both structure types, making changes as required so the correct reparation or operators are used.

## V. Evaluation

We evaluated our fuzzer on a development version of *htls*, a C-based embedded TLS 1.3 implementation – developed at the Helsinki System Security Laboratory of Huawei Technologies – that is designed especially for constrained environments such as Trusted Execution Environments [21]. In our earlier work, we have used *htls* to implement a trusted channel protocol [22] and secure enclave migration [23]. The *htls* code is currently in the process of being open sourced. We fuzzed both the TLS handshake messages and the X.509 certificates transmitted as part of the TLS handshake.

### A. Results

In this section we briefly discuss the bugs that were found in *htls* with eGMT-Fuzz. We identified a total of 11 distinct bugs with varying severity, all of which were promptly fixed when reported. Our fuzzing campaign thus contributed significantly to the quality of htls, proving the efficacy of eGMT-Fuzz. A summary of the bugs found and which operator was used to find them is shown in Table IV. In the table, eGMT refers to our basic TLS/TPL fuzzer and ASN.1 refers to its ASN.1/DER extension.

*a) Application crash with wrong TLV length.:* We found that when the ASN.1 parser in *htls* received a structure with an invalid TLV length, the parser failed and aborted the execution, causing an application crash. If left unfixed, this bug could have potentially been abused by an attacker in Denial of service (DoS) attacks. The fix consisted of checking the length of the remaining bytes in memory for the received object before attempting to read the bytes indicated by the TLV length.

*b) NULL pointer dereference in Certificate parse.:* A NULL pointer dereference was triggered when *htls* attempted to read an inconsistent X.509 certificate. The invalid certificate had an incorrect length field in the last BIT STRING of the object. This value normally contains the signature of the certificate that the application needs to verify in order to validate that the peer does possess the corresponding private key for the sent certificate. Before reading this value, *htls* initializes a pointer to NULL, which will hold the address of the signature after parsing. However, when trying to read the invalid value, *htls* noticed that the lengths were inconsistent and raised a buffer error in the parsing function, causing it to

```
1: function eGMT-Fuzz(V)                                                    ▷ Fuzz the tree V
2:     while RND({true, false}) do
3:         o ← RND({O_eVoid, O_eDup, O_Zero, O_fuzz^eTrunc, O_fuzz^Int, O_fuzz^eData, O_fuzz^eApp, O_fuzz^eGen, O_fuzz^BitF, O_fuzz^Rot})   ▷ Select a random operator
4:         if o ∈ {O_Zero, O_fuzz^Int, O_fuzz^BitF, O_fuzz^Rot} then
5:             v ← RND({v ∈ V | F_o(v) = 1})                                ▷ Select a suitable node v using filters
6:         else
7:             v ← RND({v ∈ V})                                             ▷ Select any node v
8:         end if
9:         ApplyOperator(()o, v)                                            ▷ Apply operator o to node v
10:        if v ∉ {O_Zero, O_Int^fuzz, O_BitF^fuzz, O_Rot^fuzz} then
11:            Repair(()v)                                                  ▷ Call the REPAIR function
12:        end if
13:    end while
14: end function
```

Fig. 6. Randomized fuzzing function

```
1: function Repair(v)
2:     fullRepair ← RND({true, false})
3:     while v ≠⊥ do                                  ▷ Repeat until root is reached
4:         if fullRepair or RND({true, false}) then
5:             if IsAsn1Object(v) then
6:                 RepairTlvLen(v)                    ▷ Repair DER length
7:             else
8:                 RepairLocal(v)                     ▷ Repair TPL length
9:             end if
10:        end if
11:        v ← ParentOf(v)                            ▷ Go up one level
12:    end while
13: end function
```

Fig. 7. Length repair function

fail. The problem was that the execution flow continued and the application later tried to access the signature through this NULL pointer in an address that did not exist, crashing the application immediately. This bug could have also led to DoS attacks if not fixed.

*c) Buffer over-read in log print with invalid certificates.:* Another bug was found in the function responsible for parsing X.509 certificates. This time, a buffer over-read error (when a program that reads from a buffer runs over the buffer boundaries and attempts to read from adjacent memory) occurred when logging was enabled and an invalid SubjectPublicKeyInfo (SPKI) length was set. It was found that *htls* performed buffer checks for all fields in the certificate, with the exception of the SPKI field specifically. *htls* obtained the length for the SPKI from the invalid field, which was set to a very large number, and if logging was enabled, it attempted to read as many bytes as indicated by the length and print them to the logs. If an attacker had read access to the application logs, she could have abused this bug to dump large amounts of memory and obtain sensitive information, such as private keys or cookies. When logging was disabled, however, the error was not triggered.

*d) Garbage bytes after signature.:* It was found that *htls* was ignoring any additional bytes sent after an ECDSA signature, as long as the relevant bytes were correct. This behavior does not imply a high risk on itself but is still recommended to avoid as it can potentially ease signature malleability attacks.

*e) Missing ECDH public key validation.:* TLS applications using ECDH for key negotiation need to check the public keys sent by their peers to prevent invalid curve attacks [24], which have been shown to enable an attacker to obtain the corresponding private key for the given public keys. *htls* failed to perform this check and allowed any value to be used as a public key. It was found that a Man-in-the-Middle (MitM) attacker was able to modify the `ClientHello` and `ServerHello` messages to contain public keys, e.g., belonging to other curves or comprised of zero bytes only. Given that TLS 1.3 uses ephemeral keys only, the impact of this bug is not high. However, it was still recommended to fix this as there are scenarios in which public keys are reused.

*f) Buffer over-read in log print with empty messages:* Another buffer over-read error in a log printing function was found when sending an empty message in one of either: `ClientHello`, `ServerHello`, `CertificateRequest`, `Certificate` or `CertificateVerify`. If logging was enabled, the logging function would walk through all the bytes received in the message and print them. The issue was that for reading these bytes, the function received as a parameter the amount of bytes to be read, which was calculated by subtracting a pointer to the next byte to be read from a pointer to the end of the buffer. When the message received was empty (the Record header had an indicated length of zero and no payload), the pointer to the next byte was one higher than the pointer to the buffer end, and the subtraction resulted in a value of -1. This value, however, was interpreted as an unsigned integer inside the logging function, i.e., as the value 0xffffffffffffffff in hex or 18446744073709551615 in decimal for a 64-bit processor. The logging function would then attempt to read and print this amount of bytes from memory, eventually reaching a restricted memory space and crashing the application. When logging was not enabled, the application did not attempt to read the bytes in memory, but still executed a loop until the number was reached (which would require several years to finish), causing the application

to stall indefinitely. Just like before, if an attacker had access to the application logs, she could abuse this bug to dump and read secrets from memory. In addition, for this particular bug, even when logging is disabled, the bug can be abused to cause DoS by leaving the application in an unresponsive state.

*g) NULL pointer dereference in Finished message:* In a similar way as for the previous bug, sending an empty `Finished` message caused *htls* to crash. *htls* again initialized a variable to NULL that would later hold the value of the verification signature for the handshake, but since the signature did not exist in the received message, the reading function failed and the signature value remained as a NULL pointer. The execution flow then continued and, later, *htls* tried performing a $memcpy$ operation with the variable to attempt a signature validation, however, this operation failed since the address in the pointer was invalid. Given that this also caused an application crashed, it could have been abused in DoS attacks just like previous bugs.

*h) Non-zero compression method in ClientHello:* TLS 1.3 does not support compression methods as was supported in previous versions. However, it still includes this field in the `ClientHello` and `ServerHello` messages for compatibility reasons. The RFC [1] states that any parties that are negotiating a TLS 1.3 session must leave this field empty and reject any messages that are not empty to prevent compression attacks. It was found that *htls* accepted messages regardless of whether this field was empty or not, which was considered to be a minor bug.

*i) More than one extension of the same type:* Similarly, the RFC indicates that any `ClientHello` and `ServerHello` messages should not include more than one extension of the same type, with the purpose of creating an additional layer of protection against downgrade attacks. *htls* accepted messages with more than one extension of the same type and, even if this bug was not considered to cause any security impact, it was still recommended to fix it.

*j) Missing signature_algorithms extension:* Just like for the previous bug, the RFC states that servers that receive a `ClientHello` message that does not include a `signature_algorithms` extension when using certificate authentication must abort the session. This extension indicates which signature algorithms are supported for verifying certificates and if manipulation is possible, it could arguably help to perform downgrade attacks. *htls* failed to abort the session when this extension was missing.

*k) EncryptedExtensions can be sent unencrypted:* The RFC specifies that after the `ServerHello` message has been sent, all the following messages should be sent encrypted, given that at this point both parties have enough information to derive the cryptographic keys. It was found that *htls* received and parsed an unencrypted `EncryptedExtensions` message, failing to comply with the RFC.

TABLE IV. SUMMARY OF BUGS FOUND

| Bug | Found by | Operator |
|---|---|---|
| Application crash with wrong TLV length | eGMT+ASN.1 | BitFlipOp |
| NULL pointer dereference in Certificate parse | eGMT | BitFlipOp |
| Buffer over-read in log print with invalid certificates | eGMT | BitFlipOp |
| Garbage bytes after signature | eGMT+ASN.1 | eDupOp |
| Missing ECDH public key validation | eGMT | ZeroOp |
| Buffer over-read in log print with empty messages | eGMT | eVoidOp |
| NULL pointer dereference in Finished message | eGMT | eTruncFuzzOp |
| Non-zero compression method in ClientHello | eGMT | eFuzzDataOp |
| More than one extension of the same type | eGMT | eDupOp |
| Missing signature algorithms extension | eGMT | eVoidOp |
| EncryptedExtensions can be sent unencrypted | eGMT | eAppFuzzOp |

## VI. RELATED WORK

Our work is an extension of the Walz-Sikora fuzzer [2]. The main advantage of our approach is the ability to deeply fuzz the entire TLS handshake, including encrypted and integrity-protected messages. Essentially, we introduce the "crypto filter" proposed but not implemented by Walz and Sikora. We improved and extended the original GMT operators and showed how the GMT concept can be applied to non-TLS use cases – demonstrated by fuzzing ASN.1 DER encoded signatures and X.509 certificates. A big difference between our work and the Walz-Sikora fuzzer is that we do not use differential testing; our method requires only access to the implementation under test and no further TLS libraries.

In addition to Walz and Sikora, format-aware fuzzing based on syntax tree mutation has been proposed independently by other researchers. One line of work (e.g. [25], [13]) focused on fuzzing textual formats such as Javascript and XML, targeting language parsers and interpreters. AFLSmart [11] also uses format-aware tree-based fuzzing, but is more geared towards hierarchically structured binary formats, such as WAV files. In contrast, our methodology is designed for fuzzing the binary on-the-wire format of interactive cryptographic protocols; in our proof-of-concept, we targeted TLS implementations and X.509 certificate processing code. Prior work for TLS mostly uses greybox fuzzing, whereas our method is black box and requires no access to the PUT's source code.

Like us, Zhao et al. [26] include crypto capabilities in their TLS fuzzer. Instead of a GMT-like tree structure, they propose the concept of message tuples. The authors identify 12 message tuples and, based on analysis of prior vulnerabilities, design domain-specific mutators for them. In contrast to our work, Zhao et al. focus on the earlier TLS 1.2 protocol and, like Walz and Sikora, rely on differential fuzzing to discover issues.

Similar to us, Hu et al. [3] also use the Walz-Sikora fuzzer as their starting point. However, they do not invent new GMT

operators, but rely on the original ones described by Walz and Sikora. As their main contribution, Hu et al. integrate the format-aware GMT-based fuzzer into four open source fuzzing frameworks: AFLNet, NSFuzz, StateAFL and AFLnwe. The authors show that the GMT approach provides a significant improvement over the baseline fuzzers. For example, the proportion of interesting test cases generated with the GMT fuzzer was on the average 7.65 percent higher. Although we did not perform an explicit comparison with conventional fuzzers, we believe our results are in line with these findings. The authors focus on fuzzing the TLS server, while our eGMT fuzzer works equally well for the TLS client too.

## VII. Conclusions and further work

Interactive protocols are difficult to fuzz due to lengthy message sequences and dependencies between messages. Cryptographic protocols are harder still, as they require the fuzzed input to pass integrity validation and freshness checks to achieve deep fuzzing. To address these challenges, we presented eGMT-Fuzz – a syntax-aware black box fuzzer. Our approach is based on syntax tree mutation with a pluggable protocol-specific component for cryptographic computations such as TLS termination.

With eGMT-Fuzz, we discovered 11 distinct bugs in the development version of *htls*, an embedded, C-based TLS 1.3 library. Most of these would have been difficult to detect with traditional blackbox fuzzers based on format-breaking random mutations. While direct comparison against other format-aware fuzzers was left for further work, we believe that our results demonstrate the validity of our approach.

Based on the results we obtained with eGMT-Fuzz and further fuzz testing methodologies described in the related Master's thesis [27], we believe that embedded implementations written in C have high probabilities of including bugs that can escalate to vulnerabilities. We recommended to perform fuzz testing at different stages of the development process and to follow secure coding practices such as CERT C [28] and MISRA-C [29]. Examples of useful coding guidelines specific to C include bounds checking when manipulating memory buffers and making sure that pointers are valid before use.

In the future, we plan to continue developing new eGMT operators, as we believe there is still much room for improvement. New operators could be, for example, designed based the ideas of Zhao et al. [26]. Furthermore, we intend to apply eGMT-Fuzz to new domains such as confidential computing and plan to fuzz more TLS implementations such as OpenSSL. Fuzzing remote attestation protocols [30] would also be an interesting test target, as this domain has not yet been subjected to much fuzzing and, with some extra work, eGMT-Fuzz should be well-suited to common attestation formats such as COSE and CBOR [31]. Ideal would be if eGMT-Fuzz could be developed into an open-source, developer-friendly tool

We conjecture that, despite the significant advances of recent years, the quest for fuzzing cryptographic protocols has only just begun – much remains to be discovered!

## References


[1] E. Rescorla, "The Transport Layer Security (TLS) Protocol version 1.3," RFC 8446, Aug. 2018.

[2] A. Walz and A. Sikora, "Exploiting dissent: Towards fuzzing-based differential black-box testing of TLS implementations," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, pp. 278–291, Mar. 2020.

[3] F. Hu, J. Ji, H. Shu, Z. Li, T. Liu, and C. Zhang, "Formatted stateful greybox fuzzing of TLS server," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 151–160.

[4] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[5] B. P. Miller, M. Zhang, and E. R. Heymann, "The relevance of classic fuzz testing: Have we solved this one?" *IEEE Transactions on Software Engineering*, pp. 1–1, Dec. 2020.

[6] J. W. Duran and S. Ntafos, "A report on random testing," *ICSE*, vol. 81, pp. 179–183, 1981.

[7] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, pp. 1199–1218, Sep. 2018.

[8] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, pp. 70–76, Feb. 2020.

[9] H. Zawawy and J. Bottarini, "Android goes all-in on fuzzing," urlhttps://security.googleblog.com/2023/08/android-goes-all-in-on-fuzzing.html, 2023.

[10] P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *NDSS Symposium 2008*. San Diego, USA: The Internet Society, 2008.

[11] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, pp. 1980–1997, 2021.

[12] R. Guo, "MongoDB's JavaScript fuzzer," *Communications of the ACM*, vol. 60, pp. 43–47, May 2017.

[13] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[14] C. Boyd, A. Mathuria, and D. Stebila, *Protocols for Authentication and Key Establishment*, 2nd ed. Berlin, Germany: Springer Verlag GmbH, 2020.

[15] J. Larmouth, *ASN.1 Complete*. USA: Morgan Kaufmann Academic Press, 2000.

[16] C. Research, "Sec 1: Elliptic curve cryptography, version 2.0," May 2009.

[17] Y. Pan, W. Lin, Y. He, and Y. Zhu, "Coverage-guided differential testing of TLS implementations based on syntax mutation," *PloS one*, vol. 17, no. 1, p. e0262176, 2022.

[18] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *CCS'16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Oct. 2016, pp. 1492–1504.

[19] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 615–632.

[20] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, Mass.: Addison-Wesley, 1998.

[21] L. Gunn, N. Asokan, J.-E. Ekberg, H. Liljestrand, V. Nayani, and T. Nyman, "Hardware platform security for mobile devices," *Foundations and Trends in Privacy and Security*, vol. 3, pp. 214–394, Jun. 2022.

[22] A. Niemi, V. A. B. Bop, and J.-E. Ekberg, "Trusted Sockets Layer: A TLS 1.3 based trusted channel protocol," in *Secure IT Systems: 26th Nordic Conference, NordSec 2021*, ser. Lecture Notes in Computer Science, N. Tuveri, Ed. Cham: Springer International Publishing, 2021, pp. 175–191.

[23] V. A. B. Pop, A. Niemi, V. Manea, A. Rusanen, and J.-E. Ekberg, "Towards securely migrating WebAssembly enclaves," in *EuroSec '22: Proceedings of the 15th European Workshop on Systems Security*. New York, USA: ACM, Apr. 2022, pp. 43–49.

[24] T. Jager, J. Schwenk, and J. Somorovsky, "Practical invalid curve attacks on TLS-ECDH," in *Computer Security – ESORICS 2015*. Springer International Publishing, 2015, pp. 407–425.

[25] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li, "Secure live migration of SGX enclaves on untrusted cloud," in *Proceedings of the 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, Jun. 2017, pp. 225–236.

[26] 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 *Proceedings of the 2024 IEEE Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE, 2024, pp. 918–928.

[27] A. Lomeli, "Security testing of embedded TLS implementations," Master's thesis, Aalto University, 2022.

[28] R. C. Seacord, *Secure Coding in C and C++*, 2nd ed. USA: Addison-Wesley Professional, 2013.

[29] MIRA Ltd, *MISRA-C:2004 Guidelines for the use of the C language in Critical Systems*, MIRA Std., Oct. 2004. [Online]. Available: www.misra.org.uk

[30] A. Niemi, S. Sovio, and J.-E. Ekberg, "Towards interoperable enclave attestation: Learnings from decades of academic work," in *2022 31st Conference of Open Innovations Association (FRUCT)*. IEEE, Apr. 2022, pp. 189–200.

[31] M. Moustafa, "Remote attestation for constrained relying parties," Master's thesis, Aalto University, 2023.