SADT: Syntax-Aware Differential Testing of Certificate Validation in SSL/TLS Implementations

Lili Quan
College of Intelligence and Computing, Tianjin University
China

Qianyu Guo*
College of Intelligence and Computing, Tianjin University
China

Hongxu Chen
School of Computer Science and Engineering, Nanyang Technological University, Singapore

Xiaofei Xie†
School of Computer Science and Engineering, Nanyang Technological University, Singapore

Xiaohong Li
College of Intelligence and Computing, Tianjin University
China

Yang Liu
School of Computer Science and Engineering, Nanyang Technological University, Singapore

Jing Hu†
College of Intelligence and Computing, Tianjin University
China

ABSTRACT

The security assurance of SSL/TLS critically depends on the correct validation of X.509 certificates. Therefore, it is important to check whether a certificate is correctly validated by the SSL/TLS implementations. Although differential testing has been proven to be effective in finding semantic bugs, it still suffers from the following limitations: (1) The syntax of test cases cannot be correctly guaranteed. (2) Current test cases are not diverse enough to cover more implementation behaviours. This paper tackles these problems by introducing SADT, a novel syntax-aware differential testing framework for evaluating the certificate validation process in SSL/TLS implementations. We first propose a tree-based mutation strategy to ensure that the generated certificates are syntactically correct, and then diversify the certificates by sharing interesting test cases among all target SSL/TLS implementations. Such generated certificates are more likely to trigger discrepancies among SSL/TLS implementations, which may indicate some potential bugs.

To evaluate the effectiveness of our approach, we applied SADT on testing 6 widely used SSL/TLS implementations, compared with the state-of-the-art fuzzing technique (i.e., AFL) and two differential testing techniques (i.e., NEZHA and RFCcert). The results show that SADT outperforms other techniques in generating discrepancies. In total, 64 unique discrepancies were discovered by SADT, and 13 of them have been confirmed as bugs or fixed by the developers.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Differential testing, Certificate validation, SSL/TLS Implementation

ACM Reference Format:
Lili Quan, Qianyu Guo, Hongxu Chen, Xiaofei Xie, Xiaohong Li, Yang Liu, and Jing Hu. 2020. SADT: Syntax-Aware Differential Testing of Certificate Validation in SSL/TLS Implementations. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416552

1 INTRODUCTION

The Secure Sockets Layer (SSL) [4] and its descendant Transport Layer Security (TLS) [44] protocols are the foundations of internet security. For example, Hypertext Transfer Protocol Secure (HTTPS) [45] uses them to provide trusted authentication and secure communication. SSL/TLS authenticates a server or client by validating the X.509 certificate during the handshake phase to ensure a secure connection. The authentication on a server means that the client validates the certificate presented by the server to determine whether it is a genuine communication server. The authentication on a client is also the similar way. After the server/client is authenticated, subsequent communications can be launched. Therefore, it is critically important to ensure the correctness of certificate validation by SSL/TLS implementations.

A communication may be dangerous if an invalid certificate sent by malicious server/client is falsely accepted. Similarly, a normal communication requirement may be denied if a valid certificate sent by benign server/client is falsely rejected. However, the certificate validation is a complex process involving many aspects, such as checking multiple fields (e.g., *validity*, *public key*, *extensions*) in a certificate and verifying each certificate along the certificate chain. Existing SSL/TLS implementations conduct the validation

[*] Qianyu Guo contributes equally with Lili Quan to this paper and is the co-first author.
[†] Xiaofei Xie (xfxie@ntu.edu.sg) and Jing Hu (mavis_huhu@tju.edu.cn) are the corresponding authors.

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416552

conforming to the specifications in Request For Comments (RFCs) (i.e., RFC 2527 [13], RFC 5246 [46], RFC 5878 [5], RFC 5280 [14], RFC 6101 [16], RFC 6818 [58], and RFC 6125 [47]). However, the developers may have different understandings on these specifications, which may further lead to the incorrect implementations. This makes the certificate validation remain as a weak part in the whole network ecosystem. Therefore, a systematic testing technique is still in urgent need for evaluating the quality of certificate validation of SSL/TLS implementations.

Recently, some approaches have been proposed to evaluate the SSL/TLS certificate validation process [6–8, 42] based on the differential testing technique [31]. The basic idea is to cross-check the behaviours among multiple SSL/TLS implementations. If one implementation accepts an input while another rejects it, the discrepancy occurs, which can be considered as a potential bug. However, existing differential testing based techniques still suffer from the following three limitations: 1) Existing techniques are inefficient because they heavily rely on a large number of certificates for constructing a well-behaved corpus to discover semantic bugs. For example, Frankencert [6] finds only 9 unique discrepancies among 15 SSL/TLS implementations using 243,246 seed certificates as inputs. 2) The diversity of generated certificates is not enough to cover all validation behaviours in SSL/TLS implementations, which may miss some corner cases. A typical example is RFCcert [8], which aims at generating certificates that violate the extracted rules. However, the rules may be incomplete, and many certificates that satisfy the rules but may reveal bugs are missed. 3) Some mutation-based differential testing techniques cannot guarantee that the generated certificates are syntactically correct. Such invalid certificates are more likely to be filtered, making the testing ineffective. For instance, NEZHA [42] directly mutates the certificate files, regardless of the structured X.509 syntax, and generates enormous amount of test cases that are syntactically incorrect (e.g., 97.52% syntactically incorrect certificates after 100,000 mutations in our experiments).

To tackle these problems, we propose SADT, a novel syntax-aware differential testing based framework for detecting bugs in the certificate validation process of SSL/TLS implementations. The main difference between our approach and previous work is that the certificates generated by SADT are more likely to trigger discrepancies, meanwhile the syntax is ensured to be correct. Specifically, a *tree-based mutation* strategy is first proposed to guarantee the syntax correctness of generated certificates. Then a global coverage on all targeted implementations is introduced to guide the selection of the certificate. Finally, such generated certificates are employed for the cross validation in different implementations. In order to evaluate the performance of SADT, we design experiments on 6 SSL/TLS implementations to answer the following research questions:

RQ1 : How effective is SADT in discovering discrepancies?
RQ2 : How does SADT perform compared to the state-of-the-art differential testing frameworks?
RQ3 : How does SADT perform compared to the state-of-the-art fuzzing technique (i.e., AFL)?

In summary, this paper makes the following main contributions:

- *Tree-based mutation.* We introduce a novel *tree-based mutation* strategy which parses the certificate into a tree according to the X.509 syntax and mutates the nodes on it to obtain new certificates. This method can guarantee syntactical correctness of the generated certificates and it may be applicable in the generation of other syntactically complex inputs.
- *The global coverage guidance.* We propose the *global coverage guidance* by sharing interesting test cases across all tested implementations to diversify the certificate generation that are more likely to trigger discrepancies.
- *Implementation and evaluation.* We implement SADT by extending AFL and compare against the state-of-the-art differential testing techniques (i.e., NEZHA and RFCcert) on 6 popular SSL/TLS implementations. The results show that SADT improves the capability of bug detection in these SSL/TLS implementations.
- *Community feedback.* We have reported the bugs found by SADT. To date, 13 bugs have been confirmed or fixed by developers.

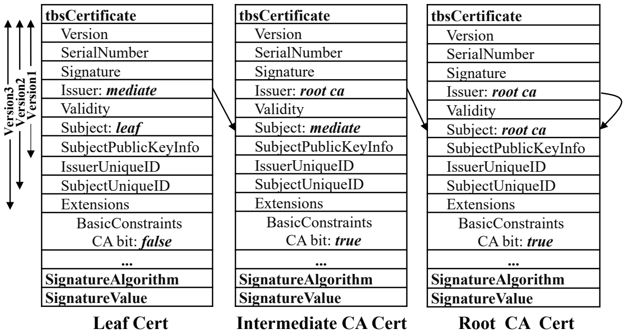


Figure 1: General structure of a certificate chain.

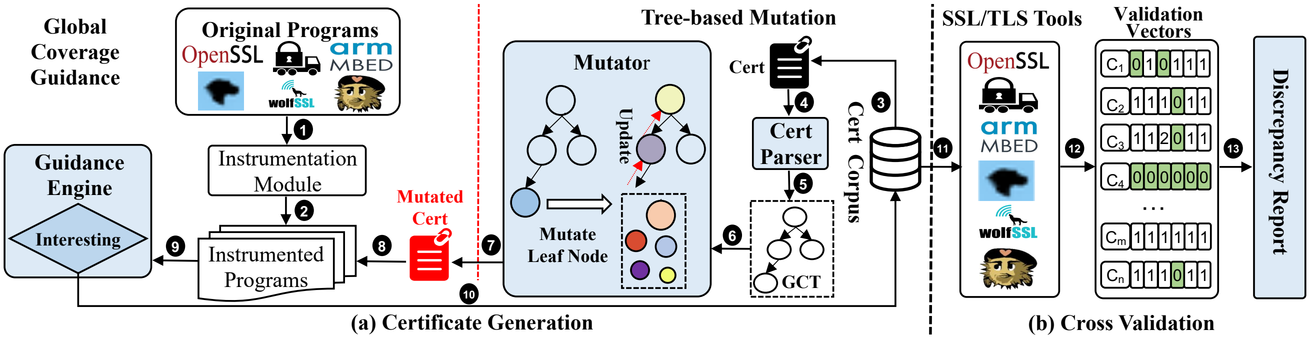
The remainder of this paper is organized as follows. Section 2 briefly introduces X.509 certificate and certificate validation. Section 3 presents the technical details of our approach including the certificate generation and cross validation. Section 4 shows the effectiveness and performance of SADT. Related work is discussed in Section 5 and this paper is concluded in Section 6.

2 BACKGROUND

2.1 X.509 Certificate

The X.509 certificate is a signed data structure that binds a public key to a person, computer, or organization and it is used in many Internet protocols, including SSL/TLS. However, it is also complex in structure and syntax. As shown in Figure 1, each certificate consists of a sequence of three required fields: (1) *tbsCertificate*, (2) *SignatureAlgorithm*, and (3) *SignatureValue.* The first part *tbsCertificate* contains a subject, an issuer and other basic information. Compared with the version-1 certificate, the version-2 certificate adds the *SubjectUniqueID* (Subject Unique Identifiers) and *IssuerUniqueID* (Issuer Unique Identifiers) fields. In addition, *Extensions* are added to the version-3 certificate. The second part *SignatureAlgorithm* contains the identifier of the signature algorithm used by Certificate Authority (CA) to sign this certificate. As the third part, *SignatureValue* records the digital signature computed upon *tbsCertificate.*

In Public Key Infrastructure (PKI), a certificate is usually organized into a certificate chain together with it's issuers. Figure 1 shows the general structure of a certificate chain. In general, each



(a) Certificate Generation (b) Cross Validation

Figure 2: Overview of SADT

certificate chain starts with a leaf certificate followed by all the issuers and each certificate in it is signed by the next certificate and the last certificate (i.e., the root certificate) is a trusted self-signed CA certificate. Moreover, the *Issuer* of a certificate is equal to the *Subject* of the next certificate. More details about X.509 certificate can be found in RFC 5280 [14].

The X.509 certificate is encoded by the Distinguished Encoding Rules (DER) [32]. The transfer syntax used by DER always follows the format $\langle Tag, Length, Value \rangle$ that is usually referred to as a TLV triplet in which each field (T, L, or V) contains one or more bytes. The *Tag* specifies the type of the data structure. For example, the *Tag* `0x02` represents the current data type is *Integer*, and the *Tag* `0x06` represents the current data type is *Object Identifier*. The *Length* specifies the number of bytes of *Value* field, and the *Value* stores data content. According to DER transfer syntax, basic and string types are encoded by using primitive forms, while constructed types are encoded in a constructed form. Note that the *Value* field is triplet if the data type is constructed form as shown in Figure 3.

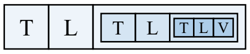


Figure 3: The DER data structure

2.2 Certificate Validation

Certificate validation is the key of authentication since it checks the genuineness and validity of certificates. It usually requires two inputs: the trusted CA certificates and a certificate chain to be validated. During the validation process, SSL/TLS implementations first check whether the content of the leaf certificate is valid at the current time, including checking the validity period and basic constraints of other fields. Then they check whether the certificate is issued by a valid CA through checking the validity of the next certificate (issuer certificate) in the certificate chain, and validating the *signature value* by the signature algorithm. This process continues along the certificate chain until the certificate be checked appears in the trusted certificate set. A certificate will be accepted if and only if all of checks mentioned above are passed. Otherwise it is rejected. More information about certificate validation can be found in RFC 5246 [46] and RFC 6125 [47].

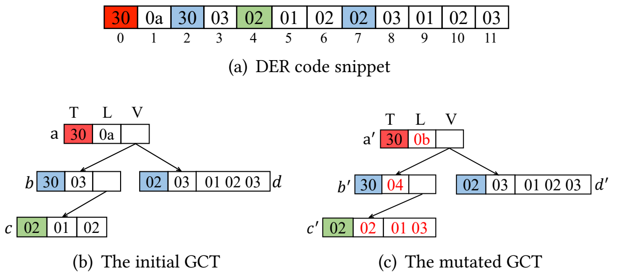
3 APPROACH

This section presents the technical details of our approach. We introduce the overview of SADT, and then discuss the algorithms of certificate generation and cross validation.

3.1 Overview

Figure 2 presents the overview of SADT, which consists of two phases: the certificate generation (Section 3.2) and the cross validation (Section 3.3). In the phase of certificate generation, new test cases (i.e., certificates) are generated with the help of *tree-based mutation* and *global coverage guidance*. Specifically, before starting to generate certificates, all of the target SSL/TLS programs are instrumented by the instrumentation module (see ❶), so that the code coverage information can be obtained in execution (see ❷). Next, SADT randomly selects a seed certificate from the certificate corpus (see ❸), which will be first parsed from a nested TLV-based DER format (Section 2.1) into a tree-based structure by the cert parser (Section 3.2.1, see ❹ and ❺). Such tree-based structure is also known as *Generic Certificate Tree (GCT)* in this paper. Then SADT conducts the tree-based mutation by randomly changing the leaf nodes and recursively updating the tree (Section 3.2.2, see ❻). The mutated tree is then parsed back to a DER-format certificate (see ❼). Afterwards, we run the generated certificate on those instrumented programs in parallel, with the support of a global guidance engine (Section 3.2.3, see ❽). Note that, the guidance engine is designed here to cover the code paths of all SSL/TLS tools as many as possible. It determines whether the mutated certificate is able to trigger new code coverage in any of the targeted SSL/TLS tools (see ❾). If yes, SADT adds the mutated certificate into the certificate corpus for further iteration; Otherwise, SADT discards it (see ❿). We repeatedly conduct the aforementioned steps for multiple times (i.e., a time budget) to obtain an extensive certificate corpus.

The certificate corpus is further deployed for cross validation (Section 3.3) when the certificate generation phase is completed. In this phase, each certificate in the corpus is fed to all SSL/TLS tools in parallel (see ⓫), and the output vector for each certificate is recorded (see ⓬). As shown in Figure 2b, an output vector consists of a certificate name (C_N) and the validation results on each implementation, where 0 means "accept" the certificate and the non-zero number means "reject" it with different return values. We get a large number of validation vectors, corresponding to individual



(a) DER code snippet

(b) The initial GCT

(c) The mutated GCT

Figure 4: The examples of data structure for DER and GCT.

test cases. SADT checks these vectors to identify whether there are discrepancies. A discrepancy is found when there are at least two implementations exhibit different outputs. Finally, we summarize the discrepancy report for further bug confirmation (see ⓭).

3.2 Certificate Generation

The performance of software testing largely depends on the quality of test cases, especially when the target program requires highly-structured inputs. Specific to SSL/TLS validation, we need a large number of representative certificates as test cases. However, as mentioned earlier, the certificates generated by previous work suffer from various limitations such as the syntactical incorrectness and high redundancy. SADT aims to address such problems by introducing two novel solutions: 1) a tree-based mutation strategy for ensuring the syntactical correctness during certificate generation, and 2) a global coverage guidance for diversified test case selection.

3.2.1 Certificate Parsing. To obtain high-quality certificates that conform to the strict syntax requirements in mutation, we need a deep understanding on the certificate composition. In other words, given a certificate, we need to address the following two concerns: how to precisely locate each field and how to extract the field value. As described in Section 2.1, the certificate is composed of a nested $\langle Tag, Length, Value\rangle$ triplet, a.k.a. TLV triplet, encoded by the DER. The structure is *nested* in that when we get a field *value* in a TLV triplet, the value itself may be another nested structure, which needs to be recursively obtained. This is similar to traversing values on the tree. Therefore, the certificate can be represented as a tree-based structure so as to facilitate the field location and value acquisition.

The *Cert Parser* is introduced for parsing the DER format certificate into a tree-based structure, which we call the *Generic Certificate Tree (GCT)* in this paper. Specifically, the GCT is a tree with multiple typed nodes, where each node is a TLV triplet, representing a particular certificate component. It should be noted that, each subsequent node next to the root node represents a certificate field, whose value is recursively determined by its sub-tree. As to the bottom of tree, the leaf node represents an atomic content which can be directly translated by DER (e.g., an integer number).

Figure 4 shows an example converting DER to GCT. Consider the DER snippet in Figure 4a, it is a nested TLV triplet, where the root tag, intermediate tags and end tag are marked as the red box (i.e., *30*), blue boxes (i.e., *30* and *02*), and green box (i.e., *02*), respectively. Specifically, for the root triplet, the tag *30* means it is a *SEQUENCE*. The length *0a* (a hexadecimal number) indicates that the values of this *SEQUENCE* occupy the following 10 bytes, ranging from the index *2* to index *11*. Similarly, we can identify the two intermediate triplets (i.e., the indices *2-6* and *7-11*, respectively) and the end triplet (i.e., the indices *4-6*). Note that, the value of end triplet is a one-byte integer number *02* which cannot be further split. Based on above observations, we can build the GCT by linking the containment relationship between each triplet. As shown in Figure 4b, node a and node c correspond to the root triplet and end triplet in Figure 4a, while node b and node d correspond to the two intermediate triplets, respectively.

Algorithm 1 presents our certificate parsing method, which takes a DER-format certificate (i.e., D) as input, and outputs a GCT (i.e., $\mathcal{T}$) corresponding to D. Since the certificate is a nested TLV structure, we can first extract the root *tag* (i.e., T), *length* (i.e., L), and *value* (i.e., V) from D (Lines 2 to 4). Then we construct an initial node for GCT (Line 5), which also consists of three parts, corresponding to the *tag*, *length*, and *value* in the TLV triplet, respectively (see Figure 4b). We directly assign the obtained tag and length to this node (Line 6). When it comes to the node value, we adopts different assignment solutions depending on the structure type (i.e., node tag). According to the DER encoding specification, if the tag belongs to the primitive types (Line 7), such as the *Bit String* (i.e., *03*) and *Integer* (i.e., *02*), it means we get an atomic TLV triplet. Thus we consider the current node as a leaf node, and directly assign the extracted *value* V to the value part of this node (Line 8). Otherwise, it means we get a nested TLV triplet, where the *value* V can be further split. Therefore, we consider the current node as an root or intermediate node, which needs to recursively conduct the same procedure for building the child nodes under its value part (Line 13). Note that, for the obtained *value* V, it may be a wrapper of multiple subsequent TLV triplets, thus we need to traverse all sub-TLV triplets on V (Lines 10 to 16). As a result, we will build multiple sub-trees under the current node. Finally, after recursively assigning or linking nodes based on the DER triplet, we can build a hierarchical structure, a.k.a., the GCT.

3.2.2 Tree-based Mutation. We perform certificate mutation based on the transformed GCT. To ensure the generated certificates are syntactically correct, we apply a bottom-up mutation strategy on GCT, which can be divided into two phases: 1) the leaf selection, and 2) the backtracking repair.

Leaf Selection. At the beginning, we first need to select a target component in certificate as the entry for mutation. With the help of GCT, we can easily obtain any component by traversing along the tree. Theoretically, each component can be considered as the candidate for mutation. However, as mentioned earlier, for some component TLV triplets in a certificate, the value part itself may be another sub-TLV triplet. As a result, a direct mutation on the value part of such triplets will inevitably destroy the certificate syntax, generating a large number of low-quality certificates. With respect to GCT, such triplets can be mapped to the intermediate nodes. By contrast, the leaf node on GCT is the atomic triplet that cannot be further split, making it syntax-free to change the value. Therefore, we select the leaf nodes as the mutation entry. Specifically, we recursively apply the Breadth-First Search (BFS) strategy on the GCT for randomly localizing a leaf node to conduct the mutation.

Backtracking Repair. Given a leaf node on GCT, we then apply the built-in mutation strategies (e.g., bitflip) of AFL to change its

Algorithm 1: Certificate Parsing

```
Input  : $D = \langle T, L, V \rangle$: An DER certificate
Output : $\mathcal{T}$: A GCT corresponding to $D$
Function certParse($D$)
    $T := getTag(D)$ ;
    $L := getLength(D)$;
    $V := getValue(D)$;
    $\mathcal{T} := initNode()$;
    $\mathcal{T} \rightarrow t, \mathcal{T} \rightarrow l := assignNode(T, L)$;
    if $isAtomic(T)$ then
        $\mathcal{T} \rightarrow v := V$;                ▷ Leaf node assignment
    else
        $P := firstTLV(V)$;
        $i := 0$;
        repeat
            $\mathcal{T} \rightarrow V_i := certParse(P)$;   ▷ Assign recursively
            $P := nextTLV(V)$;
            $i := i + 1$;
        until $P = \emptyset$;
    return $\mathcal{T}$;
```

Algorithm 2: Tree-based Mutation

```
   Input  : $\mathcal{T}$: A GCT for mutation
   Output : $\mathcal{T}'$: A new GCT after mutation
1  Function mutateAndRepair($\mathcal{T}$)
2      $n := random(10)$;
3      $\mathcal{T}_{sub} := randomBFS(\mathcal{T}, n)$;
4      if $isLeaf(\mathcal{T}_{sub})$ then
5          $V' := mutate(\mathcal{T}_{sub} \rightarrow v)$;
6          $L' := len(V')$;
7          $L := \mathcal{T}_{sub} \rightarrow l$;
8          $\Delta_1 := L' - \mathcal{T}_{sub} \rightarrow l$;
9          $\Delta_2 := len(L') - len(L)$;
10         $\mathcal{T}_{sub} \rightarrow v := V'$;           ▷ Update value
11         $\mathcal{T}_{sub} \rightarrow l := L'$;           ▷ Repair length
12         return $\mathcal{T}_{sub}$, $\Delta_1 + \Delta_2$;
13     else
14         $\Delta_1 := mutateAndRepair(\mathcal{T}_{sub})$;
15         $L := \mathcal{T} \rightarrow l$;
16         $\mathcal{T} \rightarrow l := \mathcal{T} \rightarrow l + \Delta_1$;   ▷ Repair length
17         $\Delta_2 := len(\mathcal{T} \rightarrow l) - len(L)$;
18         return $\mathcal{T}$, $\Delta_1 + \Delta_2$
19 $\mathcal{T}', \_ := mutateAndRepair(\mathcal{T})$;
20 return $\mathcal{T}'$
```

value part (i.e., not the *tag* part or *length* part). Suppose the data value before and after mutation on the *value* part are V_o and V_m, respectively. It is very likely that V_m differs from V_o in both data value and data length, due to the randomness in mutation. Since the certificate is composed of a series of nested triplets, any change in the data length of the underlying triplet will inevitably affect the data length of the upper one. Therefore, we need to recursively "repair" the node *length* part along the GCT from bottom to top. In this way, we ensure that the generated certificates always conform to the syntax requirement.

Figure 4b and Figure 4c give an example on how the syntax-aware mutation is conducted along the GCT. Consider the leaf node c in initial GCT (Figure 4b), we mutate the value from *02* to *01, 03*, and get a new leaf node c', with the length of 2 bytes (Figure 4c). Since the length of the node changes from 1 to 2 during mutation, which violates the syntax restriction of DER, we need to recursively "repair" the length of upper nodes to ensure the correctness of the syntax of the certificate. Consider the intermediate nodes (i.e., b and b') in two figures, the node length is updated from *03* to *04*. The length of the root node a is further updated from *0a* to *0b*, as shown by the node a' in Figure 4c.

Algorithm 2 details the certificate mutation process, which takes a GCT $\mathcal{T}$ as input and outputs a updated GCT (i.e., $\mathcal{T}'$) after mutation on $\mathcal{T}$. Our objective is to find a leaf node as quickly as possible, so that we can start the mutation. For this purpose, we conduct a Breadth-First Search (BFS) from the root node on, and randomly select a sub-tree $\mathcal{T}_{sub}$ (Lines 2 to 3). If $\mathcal{T}_{sub}$ is a leaf node as expected, we mutate its value part (Line 5). Note that, the change of data value (i.e., $\mathcal{T}_{sub} \rightarrow v$ to V') may also result in a change on the number of bytes occupied by this value, which may further affect the number of bytes occupied by the length part of the node. Consider the mutation example in Figure 4b and Figure 4c, we mutate the value part of leaf node c from *02* to *01, 03*, resulting a length addition by 1. The current 1-byte length part in node suffices to store the updated length. However, if the value of c is mutated to a long value (e.g., a 17-byte number), then the current allocated space for the node length part is not enough to represent such a number, and needs one more byte for accommodation. Therefore, we need record these two length changes (Lines 6 to 9) and then update the value part (i.e., $\mathcal{T}_{sub} \rightarrow v$) and length part (i.e., $\mathcal{T}_{sub} \rightarrow l$) in the current node (Lines 10 to 11). The updated node and the total length change are returned (Line 12). For the cases that the selected $\mathcal{T}_{sub}$ is an intermediate node, we need to recursively apply the random BFS strategy on the GCT, until a leaf node is found (Line 14). Similarly, for each recursion, we "repair" the length part in the current node caused by its child node (Line 16) and calculate the changes in the number of bytes the node length part takes, which is raised by the repair just now (Line 17). Then the total change in byte count($\Delta_1 + \Delta_2$) and the current sub-tree (i.e., $\mathcal{T}$) are returned to upper node (Line 18). When backtracking to the root node, we get the updated GCT $\mathcal{T}'$.

It should be noted that, the subsequent nodes next to the root of tree represent the root triplet of each certificate field. Thus when applying BFS on the root node, we can identify the field name according to the random number, which facilitates the further discrepancy localization.

3.2.3 *Global Coverage Guidance.* The built-in coverage guidance engine in AFL is not suitable for differential testing, because it only maximizes the coverage of a single program during the input generation, leaving more coverage information on other programs unused, which may help to trigger discrepancies. In this work,

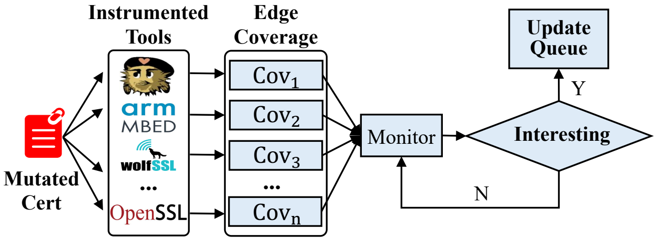


Figure 5: The workflow of global coverage guidance.

we extend the guidance engine of AFL by using global coverage information of all tested programs. Our objective is to generate the inputs that can cover as many differential behaviours as possible among multiple programs. Intuitively, a test input that is interesting for some programs, may be not interesting for other programs. The "interesting" input here refers to an input that can cover new edges or achieve a new hit count for an already-exercised edge. Therefore, sharing interesting inputs can improve the diversity of test cases, which are more likely to trigger discrepancies among different tools. The motivation can be illustrated in the following example.

Suppose A and B are two programs used for differential testing. I_1 and I_2 stand for two inputs. We also suppose that only A is fuzzed by AFL to generate test inputs, and the generated test inputs are further used for differential testing between A and B. As a result, the input I_1 covers the edges $\{A_1,A_2\}$ on A and the edges $\{B_1,B_2\}$ on B, respectively. The other input I_2 covers the edge $\{A_1\}$ on A and the edges $\{B_2,B_3\}$ on B, respectively. However, despite covering new edge on B (i.e., $\{B_3\}$), I_2 still cannot be selected for further mutation under the built-in guidance engine of AFL, because only the new coverage of A can be captured by AFL. Such situation can be addressed when we use the global coverage of both programs. Then, the input I_2 would be included for further mutation. It is similar to the case that only B is fuzzed to generate the test inputs.

The global coverage guidance engine is implemented by sharing interesting inputs among all programs involved in the differential testing. The general workflow of global coverage guidance in SADT is presented in Figure 5. When a mutation is completed, all SSL/TLS tools are executed in parallel with respect to the mutated certificate. These tools have been instrumented in advance to capture edge coverage and edge hit counts at runtime. Subsequently, the edge coverage of each program is obtained and then it is used to guide the selection of the interesting input from different SSL/TLS tools. Finally, those interesting inputs are shared among different programs by maintaining the same queue for test cases. Different SSL/TLS programs can be implemented differently and these differences can be captured by test inputs. Therefore, sharing interesting inputs can improve the diversity of test cases, which are more likely to trigger discrepancies among different tools.

Algorithm 3 presents the details about the global coverage guidance in SADT. Given an initial certificate corpus (i.e., Q), and a series of instrumented SSL/TLS tools (i.e., A), we aim to extend the corpus with a lot of mutated certificates, which expand the overall coverage when running these tools in parallel. Specifically, we initialize a coverage set of all SSL/TLS tools for test, a.k.a, the global coverage coverage (i.e., C) in Line 1. We also record the latest time that the global coverage is no longer updated (Line 2). Next, a new certificate D' is generated by mutation on a seed certificate D, which is randomly selected from the corpus (Lines 4 to 5). We run D' simultaneously on all tools and capture the global coverage by merging the coverage under each tool (Lines 6 to 8). A critical checking is then performed on D' to see whether it produces new coverage on at least one of the tools (Line 10). If yes, we consider D' as an interesting input , and add it into the corpus (Line 11). Meanwhile, we update the time when the new coverage was last generated (Line 12). We repeatedly conduct above procedures until the global coverage is no longer updated within a given period of time (Line 14). Finally, we obtain an extended corpus, with a lot of interesting certificates added.

Algorithm 3: Global Coverage Guidance

```
Input  : $A = \langle \alpha_1, \ldots, \alpha_n \rangle$: Instrumented SSL/TLS tools
         $Q$: The certificate corpus
Output : $Q$: The extended certificate corpus
Const  : $t$: A timeout threshold
1  $C := \emptyset$;
2  $T_{latest} := currentTime()$;
3  repeat
4      $D := randomSelect(Q)$;
5      $D' := mutate(D)$;
6      for $\alpha \in A$ do
7          $c = execute(\alpha, D')$;            ▹ Single coverage
8          $C := C \bigcup \{c\}$;               ▹ Global coverage
9      $T_{cur} := currentTime()$;
10     if $newCov(C)$ then
11         $Q := Q \bigcup \{D'\}$;             ▹ Add interesting cert
12         $T_{latest} := T_{cur}$;
13     $T := T_{cur} - T_{latest}$;
14 until $T > t$;
15 return $Q$;
```

3.3 Cross Validation

Differential testing [31] is a popular software testing technique that aims to detect semantic bugs, by providing the same input to a series of similar applications or different implementations of the same application, and observing discrepant behaviours in their executions. Any discrepancy in outputs of targeted applications indicates a potential bug because the specifications of these applications are theoretically identical. We propose a novel syntax-aware differential testing framework based on the certificate generation described in Section 3.2. In order to facilitate statistics and analysis of discrepancy, we define an (m+1)-dimensional validation vector $\overrightarrow{cert}$ for each tested certificate:

$$\overrightarrow{cert} =< C_N, result_1, result_2, ..., result_m >,$$

where m is the number of tested implementations, C_N is the certificate name, and the $result_i$ $(1 \leq i \leq m)$ indicates the validation result returned by the i^{th} implementation. $result_i$ is 0 if the certificate is accepted. Otherwise, it is the returned code of the rejection.

These vectors are used as metrics in the following ways.

(1) If $result_i = 0$ and $result_j \neq 0$ $(1 \leq i, j \leq m \wedge i \neq j)$, the vector is considered as a discrepancy.

(2) For a vector $< result_1, ..., result_m >_x$, if it is not equivalent to any $< result_1, ..., result_m >_y$ among k discrepancies found $(1 \leq x, y \leq k \wedge x \neq y)$, the vector is considered as a unique discrepancy.

Furthermore, in order to measure the *diversity* of certificates in a *Cert Corpus*, we define the metric [12]:

$$Diversity = \frac{|UDCert|}{|Cert|} \times 100\%,$$

where $|UDCert|$ indicates the number of unique discrepancies triggered by *Cert Corpus* and the $|Cert|$ indicates the number of certificates in *Cert Corpus*. It is clear that, given certain number of certificates, when there are more unique discrepancies, the certificates in the Cert Corpus are more diverse.

In addition, it is non-trivial to localize the root cause of each unique discrepancy due to the large number of reported discrepancies and targeted implementations. To mitigate this problem, SADT stores each mutated certificate along with its original seed certificate and records the mutated field name for each certificate, as described in Section 3.2.2. Then the root cause of a detected discrepancy can be immediately pinpointed by comparing the content of mutated field in the discrepancy-revealing certificate with the seed certificate it originates from.

4 EVALUATION

We implement SADT based on the state-of-the-art tool AFL [37] using C and Python. To evaluate the effectiveness of SADT and understand the root cause of the discrepancies and bugs, we aim to answer the research questions as mentioned in Section 1.

4.1 Experimental Setup

To evaluate SADT, the following 6 popular SSL/TLS implementations are selected: OpenSSL (v1.1.1d) [41], GnuTLS (v3.6.11) [1], MbedTLS (v2.16.3) [3], NSS (v3.48) [36], WolfSSL (v4.2.0) [57] and LibreSSL (v3.0.2) [38].

To capture discrepancies, we need to run different implementations with the same certificate and compare the outputs. Hence, we modify these implementations such that the outputs are returned, respectively. Specifically, if the certificate is accepted, the output value 0 is returned. Otherwise, we return the specific code of the rejection. We randomly download 61 real certificates from the existing websites, which are used as the initial cert corpus of SADT. Note that, all certificates in the cert corpus do not cause any discrepancies among these implementations.

All experiments are conducted on a high performance workstation, equipped with a 64-bit Ubuntu 16.04 LTS system, a 32GB RAM, and two 18-core 2.3GHz Intel Xeon E5-2699 CPUs.

4.2 RQ1: How effective is SADT in discovering discrepancies?

To detect the discrepancies between the 6 popular SSL/TLS implementations, we apply SADT to continuously generate test cases until there is no interesting test case generated within 30 minutes. Finally, SADT ran for 3 hours and generated 2305 interesting certificates (in *Cert Corpus*). Then we compare the results from 6 different implementations. We say a certificate could trigger the discrepancies if there are at least two implementations return the different results. Finally, 64 unique discrepancies are generated and the *Diversity* is 2.78%. The results show that SADT could generate diverse certificates that are more likely to trigger unique discrepancies.

Table 1: Number of unique discrepancy between each pair of SSL/TLS implementations

	OpenSSL	**GnuTLS**	**MbedTLS**	**NSS**	**WolfSSL**	**LibreSSL**
OpenSSL	-	5	3	7	8	1
GnuTLS	-	-	4	6	9	6
MbedTLS	-	-	-	5	5	4
NSS	-	-	-	-	11	8
WolfSSL	-	-	-	-	-	9
LibreSSL	-	-	-	-	-	-

The total number of unique discrepancies depends on the number of tested implementations. Intuitively, the more implementations tested, the more unique discrepancies. Then, we further show the unique discrepancies for each pair implementations. For example, suppose two output vectors <0, 1, 3, 3, 3, 3> and <0, 1, 3, 2, 1, 3> are returned from two inputs, which are regarded as the same discrepancy for the (P_1, P_2) pair. It is because the output difference is the same, i.e., (0, 1). Table 1 shows the number of unique discrepancies in each pair of implementations. Overall, there are an average of 6.06 unique discrepancies between each pair of SSL/TLS implementations. Specifically, we can observe that SADT detected the most unique discrepancies between NSS and WolfSSL (i.e., 11). Moreover, SADT detected one unique discrepancy between OpenSSL and LibreSSL although they fork the same code base. Such results demonstrate that SADT can effectively detect unique discrepancies between any two SSL/TLS implementations.

We further investigate the root cause of each discrepancy by comparing the content of fields, by mutating which (during fuzzing) the discrepancy is generated. Thus, based on checking the mutated field, the 64 unique discrepancies are classified into different categories that show the root causes. Table 2 lists the results of classification. The first column shows different fields, where SPKI, AKI, EKU, and AIA indicates the *Subject Public Key Info*, *Authority Key Identifier*, *Extended Key Usage* and *Authority Information Access*, respectively. The results show that SADT could detect more diverse discrepancies between the 6 SSL/TLS implementations. Specifically, the detected discrepancies are from 11 fields including some critical fields (e.g., Validity, Key Usage, AKI, et al.). In addition, most of the discrepancies are caused by the *Extensions*.

We have report the detected bugs found by SADT to the developers. To date, 13 bugs have been fixed or confirmed as described below. More detailed case studies are shown in Section 4.5

- *OpenSSL has confirmed 2 bugs:* (1) OpenSSL accepts a *Version* 1 certificate with *Extension* [39]. (2) OpenSSL accepts certificates including two instance of a particular extension [40].
- *GnuTLS has confirmed and fixed 5 bugs:* (1) GnuTLS accepts a certificate whose *notbefore* field is a non-digits string [18]. (2) GnuTLS can not check *Object Identifier* correctly [21]. (3) GnuTLS accepts certificates including two instance of a particular extension [19]. (4) GnuTLS can not check *Issuer* correctly [20]. (5) GnuTLS accepts *notbefore* with length 11 [22].

Table 2: The distribution of unique discrepancies found by SADT in terms of infected fields

Field Name	Version	SerialNumber	Subject	Issuer	Validity	Signature	SPKI	Extensions				Total
								KeyUsage	AKI	EKU	AIA	
Number	5	4	6	6	9	3	11	5	8	6	1	64

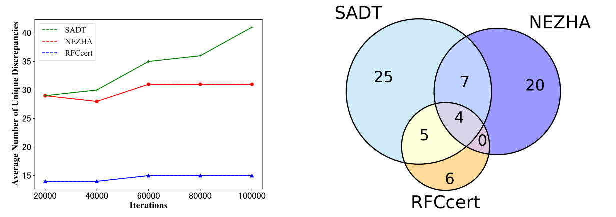
- *MbedTLS has confirmed 1 bug:* MbedTLS accepts invalid certificates whose *key identifier* of the *Authority Key Identifier* is not the same as *Subject Key Identifier* of issuer certificate [30].
- *NSS has confirmed 2 bugs:* (1) NSS accepts a *Version* 1 certificate with *Extension* [34]. (2) NSS accepts *notbefore* with length 11 [35].
- *WolfSSL has fixed 3 bugs:* (1) WolfSSL accepts a certificate with *Authority Key Identifier* that does not match issuer certificate [55]. (2) WolfSSL accepts a certificate with an invalid time format [54]. (3) WolfSSL accepts a certificate whose *Issuer* does not match the *Subject* of issuer certificate [56].

Answer to RQ1: SADT is effective in finding unique discrepancies. Specifically, SADT finds 64 unique discrepancies (including 13 confirmed bugs) across 6 SSL/TLS implementations and finds average 6.06 unique discrepancies between each pair of implementations. Moreover, the discrepancies are diverse and could cover most of fields in the certificate.

4.3 RQ2: How does SADT perform compared to state-of-the-art differential testing frameworks?

We select two state-of-the-art techniques as the baselines, i.e., RFCcert [8] and NEZHA [42], which have been shown effective in detecting discrepancies between SSL/TLS implementations. We do not select Mucert [12] and Frankencert [6] due to that RFCcert [8] has been demonstrated to be the more effective one than them. Since the source code of RFCcert is not applicable, we re-implement it according to the algorithm described in [8].

Note that, unlike SADT and NEZHA, RFCcert is a rule-based framework, which only generates certificates with the existing rules extracted from RFCs. To make a fair comparison, SADT is restricted to only mutate the certificate fields involved in the rules used by RFCcert. The same *Cert Corpus* are also provided for SADT and NEZHA. Furthermore, NEZHA has two kinds of guidance: gray-box guidance and black-box guidance. Since SADT is gray-box framework guided by global coverage guidance, the gray-box guidance of NEZHA is used in all comparison. In our experiment, 11 rules are extracted from RFC5280 [14] and these rules involve 10 common fields of X.509 certificate. We run each frameworks five times and calculate the average results for comparison. The performance of SADT, NEZHA and RFCcert are compared in terms of the number of unique discrepancies and the diversity of certificates.



(a) The number of unique discrepancies found by SADT, NEZHA and RFCcert under different iterations

(b) The distribution of unique discrepancies found by SADT, NEZHA and RFCcert under 100,000 iterations

Figure 6: The performances of SADT, NEZHA and RFCcert

4.3.1 The number of unique discrepancies. Figure 6a shows the average number of unique discrepancies discovered by SADT, NEZHA and RFCcert under different iterations. We could observe that SADT finds more unique discrepancies than NEZHA and RFCcert under the same iterations. For example, SADT finds about *1.4 times* and *4 times* unique discrepancies than NEZHA and RFCcert respectively under 100,000 iterations. Furthermore, the number of unique discrepancies found by SADT keeps increasing in different iterations while NEZHA and RFCcert stop detecting new discrepancies after 60,000 iterations. The results demonstrate that SADT outperforms NEZHA and RFCcert in finding unique discrepancies. The main reason is that NEZHA mutates the inputs without knowing the structure information of the input. RFCcert considers the syntax information and ensures the syntactical correctness of the generated certificates, but it lacks the coverage feedback. Differently, SADT considers both of the two perspectives, i.e., generating interesting test cases based on the syntactical structure and the coverage guidance, making SADT more effective.

Figure 6b shows the distribution of the unique discrepancies found by each tool under 100,000 iterations. As seen in Figure 6b, the number of unique discrepancies found by SADT, NEZHA and RFCcert are 41, 31 and 15, respectively. The results show that these tools could generate different discrepancies. For examples, 25, 20 and 6 unique discrepancies are only found by SADT, NEZHA and RFCcert, respectively, i.e., the discrepancies cannot be found by other two tools. This is due to differences in their respective methods with regards to the certificate generation. NEZHA randomly mutates certificates at the granularity of the entire certificate, thus many test cases generated by NEZHA violate the syntax of the certificate. In other words, NEZHA may find more discrepancies at the syntax parsing stage. For example, in our experiment, 97522 certificates (97.52%) generated by NEZHA are syntactically incorrect after 100,000 mutations. However, certificates generated by RFCcert have correct syntax but are not diverse enough, i.e., all of the generated certificates violate RFC specifications. However, the tree-based mutation and global coverage guidance make SADT address the above two problems, making SADT more effective.

In addition, NEZHA and RFCcert miss 73.17% and 78.05% of the discrepancies found by SADT, respectively. We also found that some of discrepancies detected by RFCcert and NEZHA are missed by SADT. This is because the current mutation strategies of SADT might be incomplete. In order to ensure that the generated certificate have the valid syntax, SADT focuses on changing the value of

Table 3: The discrepancy diversity of SADT, NEZHA, and RFCcert under different mutation iterations.

Iterations	SADT			NEZHA	RFCcert
	60,000	80,000	100,000	60,000	60,000
Uniq.	35	36	41	31	15
Certificates	1019	1125	1172	6638	60,000
Diversity(%)	3.43	3.20	3.50	0.47	0.03

the certificate field but does not change the structure of the certificate, such as adding or deleting certificate fields. Despite the fact that the mutation operators of SADT may be limited, SADT still detects more bugs than NEZHA and RFCcert due to the combination of syntax-guided and coverage-guided mutation. We plan to extend more mutation operators on SADT in the future.

4.3.2 The diversity of certificates. As shown in Figure 6a, no new unique discrepancies are discovered by NEZHA and RFCcert after 60,000 iterations, while new unique discrepancies continue to be found by SADT. Hence, the *Diversity* of certificates generated by NEZHA and RFCcert decreases as iterations increases. Table 3 shows that the *Diversity* of certificates generated by each framework and Row *Uniq.* indicates the number of unique discrepancies. As shown in Table 3, after 60,000 iterations, 1019, 6638 and 60,000 certificates are generated by SADT, NEZHA, and RFCcert, respectively, and each of them find 35, 31, and 15 unique discrepancies, respectively. Therefore, the diversity of certificates generated by SADT, NEZHA and RFCcert are 3.43% (35/1019), 0.47% (31/6638) and 0.03% (15/60000), respectively. It is obvious that the diversity of certificates generated by SADT is much greater than that of NEZHA and RFCcert. Note that, the diversity of SADT under 80,000 and 100,000 iterations are also close to that under 60,000 iterations and they are still much greater than that of NEZHA and RFCcert under 60,000 iterations. These results demonstrate the certificates generated by SADT are more diverse than NEZHA and RFCcert.

Answer to RQ2: Compared with state-of-the-art differential testing techniques, SADT is more effective to generate diverse certificates that could detect more unique discrepancies.

4.4 RQ3: How does SADT perform compared to the state-of-the-art fuzzing technique (i.e., AFL)?

SADT is implemented by extending the built-in mutation and guidance engine of AFL. Therefore, we compare the performance of SADT with the general-purpose AFL in terms of the number of unique discrepancies to further illustrate the effectiveness of our approach. Since AFL does not support differential testing, we adapt AFL for differential testing as follows: using AFL to generate certificates based on a single SSL/TLS implementation (i.e., OpenSSL) and then invoking the validation routines from 6 different implementations with these generated certificates. Moreover, to evaluate the contribution of components in SADT (i.e., the tree-based mutation and the global coverage guidance), we conduct another variant of SADT, named S-SADT, which performs the tree-based mutation but the default coverage guidance of AFL. Then we feed the same initial *Cert Corpus* (61 certificates) to SADT, S-SADT and AFL, and obtain the results. To reduce the randomness, each tool is run five times and the average results are compared.

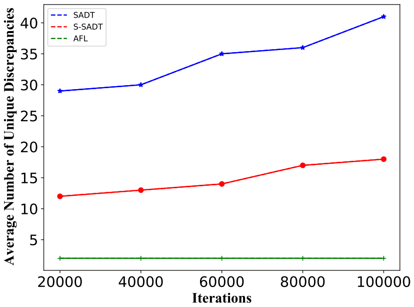


Figure 7: The number of unique discrepancies found by SADT, S-SADT and AFL under different iterations

The number of unique discrepancies discovered by SADT, S-SADT and AFL under different iterations are presented in Figure 7. It can be observed that AFL only finds 2 unique discrepancies and the number of unique discrepancies remains the same as iterations increase. Whereas, SADT and S-SADT find more unique discrepancies than AFL under same iterations and the number of unique discrepancies keeps increasing during the iterations. The number of unique discrepancies discovered by SADT is about *20 times* than that of AFL under 100,000 iterations. Consider the results between S-SADT and AFL, we found that S-SADT finds *9 times* more unique discrepancies than AFL under 100,000 iterations, which demonstrates that the tree-based mutation is effective in detecting discrepancies. Consider the results between SADT and S-SADT, we found that SADT could detect much more discrepancies (e.g., more than 2 times) than S-SADT under the same iterations, which demonstrates the effectiveness of the global coverage guidance in improving the performance of SADT. In general, SADT improves the capability of AFL to find bugs in SSL/TLS implementations.

Answer to RQ3: SADT finds about *2 times* more unique discrepancies than S-SADT. S-SADT finds *9 times* more unique discrepancies than AFL under 100,000 iterations. It is clearly that SADT and S-SADT outperform AFL in finding unique discrepancies. The results demonstrate that both of tree-based mutation and the global coverage guidance are effective in detecting unique discrepancies.

4.5 Case Studies of Bugs

To understand the root cause of the bugs reported by SADT, we manually analyze the detailed implementation and summarize the root causes of these bugs into two categories: (1) Lack of checking of corner case. The SSL/TLS implementation developers may ignore some cases that may not be clearly stated in the RFCs. For example, GnuTLS lacks some checks on whether the *Validity* value is digital string as shown in Listing 1. (2) Incorrect implementation of rules in RFCs. The developer's wrong understanding of rules may lead to the rules not being implemented correctly. The code for finding issuer on WolfSSL was implemented incorrectly as shown in Listing 2,

```
1   static  time_t utcTime2gtime(const char *ttime){
2       char xx[3];
3       int year;
4       if ( strlen (ttime)<10){
5           gnutls_assert ();
6           return (time_t)-1;
7       }
8       xx[2]=0;
9       memcpy(xx,ttime,2);
10      /* year is 0 if xx is non-digits */
11      year=atoi(xx);
12      ttime +=2;
13      if (year>49)
14          year+=1900;
15      else
16          year+=2000;
17      return time2gtime(ttime,year);
18  }
```

Listing 1: GnuTLS cannot parse validity field correctly

which is a typical example. In this section, we respectively introduce the two typical bug cases in detail.

4.5.1 GnuTLS-Incorrect validation of validity. The *Validity* (including *notBefore* and *notAfter*) of the X.509 certificate has two representations: *UTCTime (YYMMDDHHMMSSZ) and GeneralizedTime (YYYYMMDDHHMMSSZ)*, which contain 13 and 15 characters, respectively. It is obvious that the value of *notBefore* and *notAfter* cannot be a non-digital string. However, we find that GnuTLS erroneously accepts a malformed certificate whose *notBefore* or *notAfter* contain non-digital characters while other SSL/TLS implementations reject it. After manually debugging the implementation from the discrepancy, we found that GnuTLS lacks the check on whether the *Validity* value is digital string as shown in Listing 1. For example, while other implementations reject a certificate whose *notBefore* field is UTCTime *#01010101000Z* since it is a incorrect time format. However, GnuTLS incorrectly interprets the time as **Otc 10 10:10:00 1900 GMT**. This is because the function $atoi(xx)$ returns 0 when xx is non-digits (Line 11 in Listing 1). We have reported this bug to the corresponding developers and committed a Merge Request to fix it. The GnuTLS team has confirmed this bug and has fixed it in GnuTLS v3.6.12 [18].

There are no such rules like *"validity MUST be digital string"* in RFCs, which makes RFCcert unable to generate such certificates. Therefore this bug cannot be discovered by RFCcert. The result also demonstrates that SADT could be a supplement to RFCcert.

4.5.2 WolfSSL-Incorrect validation of issuer. As described in section 4.2.1.2 of RFC5280, the **Subject Key Identifier** of a CA certificate MUST match the **Authority Key Identifier** of certificates issued by the CA. However, our experiments show that WolfSSL accepts a certificate that violates this rule while other SSL/TLS implementations reject such certificate because they could not find the issuer. Through debugging, we find that the code for finding issuers in WolfSSL are not implemented correctly. As shown in Listing 2, WolfSSL will look up the issuer certificate by matching the value of *Issuer* when it could not find the corresponding issuer certificate by matching *Authority Key Identifier* (Lines 7 to 12 in Listing 2). In this case, WolfSSL will accept the certificate if the *Issuer* matches the *Subject* of the issuer certificate. WolfSSL has confirmed and fixed this bug [55], and MbedTLS has also confirmed it [30].

```
1   int ParseCertRelative (DecodedCert* cert, int type, int verify, void
        * cm){
2       ...
3   if ( verify != NO_VERIFY && type != CA_TYPE && type !=
        TRUSTED_PEER_TYPE) {
4       cert->ca = NULL;
5       #ifndef NO_SKID
6       /*CA certificate is found if one field matches the cert*/
7       if (cert->extAuthKeyIdSet)
8           cert->ca = GetCA(cm, cert->extAuthKeyId);
9       if (cert->ca == NULL && cert->extSubjKeyIdSet)
10          cert->ca = GetCA(cm, cert->extSubjKeyId);
11      if (cert->ca == NULL)
12          cert->ca = GetCAByName(cm, cert->issuerHash);
13      ...
14    }
```

Listing 2: WolfSSL cannot find issuer correctly

4.6 Threats to Validity

The selected versions of SSL/TLS implementations in our study could be a threat to validity. This work mainly focuses on the different SSL/TLS implementations. The threat could be reduced by selecting more different versions of the SSL/TLS implementations. The initial cert corpus may be a threat. We mitigate this issue by randomly selecting diverse certifications from the existing websites. Another threat would be the randomness when comparing discrepancies detection between SADT and the baselines. To mitigate this issue, we run each tool five times and calculate the average results.

5 RELATED WORK

We summarize the related work in following two aspects: the security of SSL/TLS implementations and the mutation-based testing.

5.1 Security of SSL/TLS implementations

Recently, many researches have been proposed to evaluate the security of SSL/TLS implementations. Marlinspike [29, 33] found several vulnerabilities in the certificate validation process of SSL/TLS implementations. Kaminsky et al. [23] demonstrated two new types of collision attacks against the X.509 certificate. Georgiev et al. [17] analyzed several vulnerabilities that are caused by badly designed APIs of SSL implementations (i.e., OpenSSL). Their results revealed the security risks in SSL/TLS implementations. These results motivate us to detect bugs of certificate validation in SSL/TLS implementations through automated methods.

Frankencert [6] is proposed for the first time to test certificate validation logic in SSL/TLS implementations. Chen et al. [12] further applied a guided technique to improve Frankencert (i.e., Mucert). Different from our work, Frankencert is unguided and Mucert guides certificate generation based on a single implementation instead of multiple targeted implementations. DRLgencert [7] first applied deep reinforcement learning to the automated testing of SSL/TLS implementations. It needs to extract features from a large number of certificates, which is not required in our work. TLS-Attacker [50] evaluated the security of TLS libraries by two-step fuzzing approach. Sivakorn et al. presented HVLearn [49] to find bugs in hostname verification. They are orthogonal to our work as they mainly focus on protocol level or hostname verification while SADT focuses on certificate validation process.

The most relevant work to SADT are NEZHA [42] and RFCcert [8]. Our work distinguishes from them in the following aspects: 1) RFCcert assembles certificates depending on the rules extracted from RFCs while we generate certificates in a rule-independent way, guided by the global coverage of all SSL/TLS implementations; 2) NEZHA directly mutates the certificate files regardless of the certificate syntax, while SADT leverages a tree-based mutation to generate syntactically correct certificates; and 3) we propose a more fine-grained bug localization than NEZHA.

5.2 Mutation-based Testing

In the recent years, there are many mutation-based techniques proposed for software testing, e.g., AFL [37], libFuzzer [28], FairFuzz [25], Steelix [26], Cerebro [27], Hawkeye [10], MUZZ [9], and UAFL [51], which mutate test cases with the guidance of customized domain-specific code coverage. Their guidance is ill-suitable for differential testing because it guides mutation based on a single implementation instead of multiple implementations. SlowFuzz [43] and PerfFuzz [24] generate test cases based on the resource usage. Vuzzer [48] and Angora [11] use taint analysis to identify which bytes should be mutated. They mutate test case regardless of the syntax of test case while SADT leverages a tree-based mutation to ensure mutated test case is syntactically correct. However, there are some techniques are proposed to generate syntactically correct test case. μ4SQLi [2] generates executable SQLs by applying mutation operators on valid SQLs. Superion [53] leverages grammar-aware trimming strategy and two grammar-aware mutation strategies to ensure the mutated test case is syntactically correct. Domato [15] generates test cases by specifying the syntax of HTML/CSS structure and JavaScript objects. Skyfire [52] generates well-distributed test cases by leveraging the knowledge of many existing samples. It should be noted that, above techniques are orthogonal to our work as they mainly focus on generating highly-structured inputs (such as JavaScript) instead of X.509 certificate.

6 CONCLUSION

In this paper, we design, implement and evaluate a syntax-aware differential testing framework, i.e. SADT, for testing certificate validation in SSL/TLS implementations. Specifically, the tree-based mutation and the global coverage guidance are extended on AFL to effectively mutate and diversify X.509 certificates while keeping the certificate syntactically correct. These generated certificates are leveraged to identify the discrepancies between different SSL/TLS implementations. Our experimental results demonstrate that SADT is more effective than the state-of-the-art differential testing frameworks (i.e., NEZHA and RFCcert) in detecting discrepancies and the general-purpose fuzzing technique AFL. Overall, SADT finds 64 unique certificate validation discrepancies on 6 widely used SSL/TLS implementations. In particular, 13 bugs have been confirmed or fixed by the developers.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comprehensive feedback. This work was partly supported by the National Science Foundation of China (No. 61872262, 61572349). It was also sponsored by the Singapore Ministry of Education Academic Research Fund Tier 1 (Award No. 2018-T1-002-069), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), the Singapore National Research Foundation under NCR Award Number NSOE003-0001 and NRF Investigatorship NRFI06-2020-0022.

REFERENCES

[1] Tim Rühsen,Daiki Ueno,Dmitry Baryshkov. 2020. *The GnuTLS Transport Layer Security Library.* https://www.gnutls.org

[2] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. 2014. Automated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2610384.2610403

[3] ARM Limited. 2020. *armMBED.* https://tls.mbed.org

[4] Richard Barnes, Martin Thomson, Alfredo Pironti, and Adam Langley. 2015. Deprecating Secure Sockets Layer Version 3.0. RFC 7568. https://doi.org/10.17487/RFC7568

[5] Mark Brown and Russ Housley. 2010. Transport Layer Security (TLS) Authorization Extensions. RFC 5878. https://doi.org/10.17487/RFC5878

[6] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 114–129.

[7] Chao Chen, Wenrui Diao, Yingpei Zeng, Shanqing Guo, and Chengyu Hu. 2018. DRLGENCERT: Deep Learning-based Automated Testing of Certificate Verification in SSL/TLS Implementations. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 48–58.

[8] Chu Chen, Cong Tian, Zhenhua Duan, and Liang Zhao. 2018. RFC-directed differential testing of certificate validation in SSL/TLS implementations. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 859–870.

[9] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2325–2342. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[10] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108.

[11] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725.

[12] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 793–804.

[13] Santosh Chokhani, Warwick Ford, Randy Sabett, Charles Merrill, and Stephen Wu. 1999. RFC 2527: Internet X. 509 public key infrastructure certificate policy and certification practices framework. *Internet Engineering Task Force (IETF), RFC* (1999).

[14] David Cooper, Stefan Santesson, S Farrell, Sharon Boeyen, Rusell Housley, and W Polk. 2008. RFC 5280: Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile. *IETF, May* (2008).

[15] Fratric. 2017. The great dom fuzz-off of 2017. https://googleprojectzero.blogspot.sg/2017/09/the-great-dom-fuzz-off-of-2017.html.

[16] Alan Freier, Philip Karlton, and Paul Kocher. 2011. Rfc 6101: The secure sockets layer (SSL) protocol version 3.0. *The Internet Engineering Task Force (IETF)* (2011).

[17] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 38–49.

[18] gnutls. 2019. *Gnutls accepts a certificate whose notbefore field is a non-digits string while openssl rejects such certificates.* https://gitlab.com/gnutls/gnutls/-/issues/870

[19] gnutls. 2019. *gnutls accepts certificates including two instance of a particular extension.* https://gitlab.com/gnutls/gnutls/-/issues/887

[20] gnutls. 2019. *gnutls can't check certificate issuer correctly according to RFC5280.* https://gitlab.com/gnutls/gnutls/-/issues/885

[21] gnutls. 2019. *gnutls can't check object identifier value correctly.* https://gitlab.com/gnutls/gnutls/-/issues/886

[22] gnutls. 2019. *GnuTLS3.6.7.1 cannot process validity field according to RFC5280.* https://gitlab.com/gnutls/gnutls/-/issues/864

[23] Dan Kaminsky, Meredith L Patterson, and Len Sassaman. 2010. PKI layer cake: New collision attacks against the global X. 509 infrastructure. In *International Conference on Financial Cryptography and Data Security*. Springer, 289–303.

[24] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[25] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. ACM Press. https://doi.org/10.1145/3238147.3238176

[26] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[27] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 533–544.

[28] LLVM. 2020. *libFuzzer-a library for coverage-guided fuzz testing - LLVM 3.9 documentation.* http://llvm.org/docs/LibFuzzer.html

[29] Moxie Marlinspike. 2009. More tricks for defeating SSL in practice. *Black Hat USA* (2009).

[30] mbedtls. 2019. *mbedtls2.16.3 accepts invalid certificate whose key identifier field of the authority key identifier extension is not the same as subject key identifier in issuer.* https://github.com/ARMmbed/mbedtls/issues/2954

[31] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[32] Microsoft. 2020. *Distinguished Encoding Rules.* https://docs.microsoft.com/en-us/windows/win32/seccertenroll/distinguished-encoding-rules

[33] Marlingspike Moixe. 2009. New tricks for defeating ssl in practice. In *BlackHat Conference, USA*.

[34] mozilla. 2019. *NSS accepts a version-1 certificate with extension fields.* https://bugzilla.mozilla.org/show_bug.cgi?id=1603034

[35] mozilla. 2019. *NSS UTCTime parser should reject short fields.* https://bugzilla.mozilla.org/show_bug.cgi?id=1599331

[36] Mozilla. 2020. *Network Security Services.* https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS

[37] M.Zalewski. 2020. *american fuzzy lop.* http://lcamtuf.coredump.cx/afl/

[38] OpenBSD. 2020. *LibreSSL.* https://www.libressl.org/

[39] OpenSSL. 2019. *openssl accepts a certificate with version 1 and extension fields.* https://github.com/openssl/openssl/issues/10599

[40] OpenSSL. 2019. *openssl accepts certificates including two instance of a particular extension.* https://github.com/openssl/openssl/issues/10686

[41] OpenSSL Software Foundation. 2020. *OpenSSL.* https://www.openssl.org

[42] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 615–632.

[43] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2155–2168. https://doi.org/10.1145/3133956.3134073

[44] Marsh Ray, Alfredo Pironti, Adam Langley, Karthikeyan Bhargavan, and Antoine Delignat-Lavaud. 2015. Transport Layer Security (TLS) session hash and extended master secret extension. *Transport* (2015).

[45] Eric Rescorla. 2000. HTTP Over TLS. RFC 2818. https://doi.org/10.17487/RFC2818

[46] Eric Rescorla and Tim Dierks. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. https://doi.org/10.17487/RFC5246

[47] Peter Saint-Andre and Jeff Hodges. 2011. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X. 509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). *RFC* 6125 (2011), 1–57.

[48] Vivek Jain Sanjay Rawat, Lucian Cojocar Ashish Kumar, and Herbert Bos Cristiano Giuffrida. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS Symposium 2017*.

[49] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. 2017. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 521–538.

[50] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1492–1504. https://doi.org/10.1145/2976749.2978411

[51] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *42nd International Conference on Software Engineering*. ACM.

[52] J. Wang, B. Chen, L. Wei, and Y. Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594.

[53] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2018. Superion: Grammar-Aware Greybox Fuzzing. *CoRR* abs/1812.01197 (2018). arXiv:1812.01197 http://arxiv.org/abs/1812.01197

[54] wolfssl. 2019. *wolfssl 4.0.0 accepts a certificate with an invalid time format.* https://github.com/wolfSSL/wolfssl/issues/2657

[55] wolfssl. 2019. *wolfssl-4.0.0 accepts a certificate with authority key identifier extension field that do not match issuer .* https://github.com/wolfSSL/wolfssl/issues/2659

[56] wolfssl. 2019. *wolfssl4.2.0 accepts a certificate whose issuer not matching the subject of CA certificate.* https://github.com/wolfSSL/wolfssl/issues/2680

[57] wolfSSL. 2020. *wolfSSL.* https://www.wolfssl.com/

[58] Peter E. Yee. 2013. Updates to the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 6818. https://doi.org/10.17487/RFC6818