RFC-Directed Differential Testing of Certificate Validation in SSL/TLS Implementations

Chu Chen
ICTT and ISN Laboratory, Xidian University
Xi'an, 710071, P.R. China

Cong Tian
ICTT and ISN Laboratory, Xidian University
Xi'an, 710071, P.R. China
ctian@mail.xidian.edu.cn

Zhenhua Duan
ICTT and ISN Laboratory, Xidian University
Xi'an, 710071, P.R. China
zhhduan@mail.xidian.edu.cn

Liang Zhao
ICTT and ISN Laboratory, Xidian University
Xi'an, 710071, P.R. China

ABSTRACT

Certificate validation in Secure Socket Layer or Transport Layer Security protocol (SSL/TLS) is critical to Internet security. Thus, it is significant to check whether certificate validation in SSL/TLS is correctly implemented. With this motivation, we propose a novel differential testing approach which is directed by the standard Request For Comments (RFC). First, rules of certificates are extracted automatically from RFCs. Second, low-level test cases are generated through dynamic symbolic execution. Third, high-level test cases, i.e. certificates, are assembled automatically. Finally, with the assembled certificates being test cases, certificate validations in SSL/TLS implementations are tested to reveal latent vulnerabilities or bugs. Our approach named RFCcert has the following advantages: (1) certificates of RFCcert are discrepancy-targeted since they are assembled according to standards instead of genetics; (2) with the obtained certificates, RFCcert not only reveals the invalidity of traditional differential testing but also is able to conduct testing that traditional differential testing cannot do; and (3) the supporting tool of RFCcert has been implemented and extensive experiments show that the approach is effective in finding bugs of SSL/TLS implementations.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → *Security protocols*;

KEYWORDS

Differential testing, certificate validation, SSL/TLS, dynamic symbolic execution, Request For Comments

ACM Reference Format:
Chu Chen, Cong Tian, Zhenhua Duan, and Liang Zhao. 2018. RFC-Directed Differential Testing of Certificate Validation in SSL/TLS Implementations. In *ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180226

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180226

1 INTRODUCTION

Security of Internet services is an important issue since a malicious attack could result in unpredictable consequences, even disasters. To provide trustworthy authentication and secure communication, Hypertext Transfer Protocol Secure (HTTPS) [54], as the upgrade of HTTP [19, 20], adopts Secure Socket Layer [2, 22, 62] or Transport Layer Security [4, 7, 18, 24, 34, 35, 46, 53, 55] protocol (SSL/TLS) to guarantee security. SSL/TLS authenticates a server or client by validating its certificate in the phase of handshake. As for the authentication of a server, the server sends its X.509 certificate to a client and the client validates the received certificate to determine whether it is the genuine communication server. The authentication of a client can be conducted similarly. Only if a server/client is authenticated, can subsequent communication continue. However, the communication may be dangerous if the authentication falsely accepts an invalid certificate provided by a malicious server/-client. Likewise, a reasonable requirement for communication may be denied if the authentication falsely rejects a valid certificate which is owned by a benign server/client. Therefore, it is critical to implement certificate validation correctly.

X.509 certificate validation is described in several Request For Comments (RFC) including RFCs 2527, 5246, 5878, 5280, 6101, 6818, and 6125 [7, 14, 15, 18, 22, 56, 65]. Developers' misunderstanding of these RFCs may bring in bugs during the implementations of certificate validation, and these bugs may further result in attacks, e.g. the man-in-the-middle (MITM) attack [32, 49]. So, to secure SSL/TLS implementations, it is necessary to detect bugs in certificate validation. Nevertheless, this task is a great challenge due to the following facts: (1) an X.509 certificate has a complex structure and intricate semantic constraints, (2) many hash and encryption/decryption algorithms as well as private/public keys are involved in the processes of signature and validation, and (3) certificate chains which are common in SSL/TLS are more complicated than a single certificate.

Researchers in the community have made great efforts to test certificate validation in SSL/TLS implementations. Brubaker et al. put forward the first methodology named Frankencert [8], which generates 8,127,600 new certificates by combining components of

243,246 certificates selected from the Internet. Their testing of 15 different SSL/TLS implementations yields 208 discrepancies, among which only 9 are distinct discrepancies [11]. In order to improve the efficiency of differential testing, Chen et al. propose a mutation approach named Mucert [11], which is guided by code coverage and adopts the Markov Chain Monte Carlo (MCMC) sampling method. Mucert employs 37 mutators to mutate 1,005 certificates for a certain number of times. After each round of mutation, a mutated certificate selected by MCMC sampling is reserved. With mutated certificates, 27 distinct discrepancies are found when testing 9 SSL/TLS implementations.

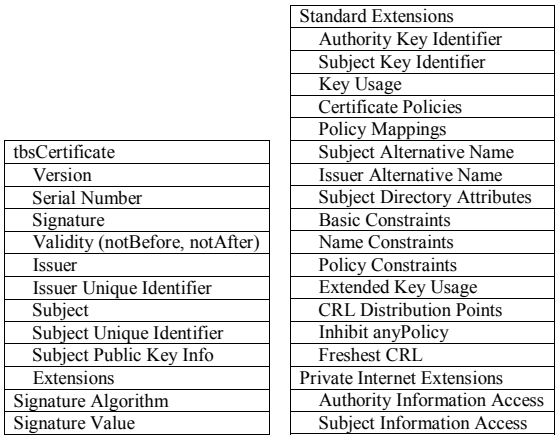
Although differential testing has been successfully applied to certificate validation of SSL/TLS implementations, there are still problems with the method. First, a discrepancy found in differential testing shows neither why it occurs nor which implementations go wrong. It is difficult for developers to fix bugs without such explicit information. Second, some of the mutated certificates used in differential testing cannot be opened by the utilities available. So, it is difficult to analyze their structures or contents. Third, it is an open question whether differential testing of certificate validation in SSL/TLS implementations is always valid.

To solve the problems above, we propose a novel differential testing approach, namely RFCcert, to detect bugs in certificate validation of SSL/TLS implementations. Different from traditional differential testing, RFCcert starts from extracting rules directly from RFCs. Then, a symbolic C program related to the rules is generated and low-level test cases, each of which represents a few requirements for components of a certificate, are produced through Dynamic Symbolic Execution (DSE) approach. Subsequently, high-level test cases, i.e. certificates, are assembled from the low-level test cases automatically. Finally, the assembled certificates are employed to test certificate validation in SSL/TLS implementations.

The advantages of our approach are summarized as follows.

(1) Certificates of RFCcert are assembled according to standards. Whenever a discrepancy is found, the certificate shows which implementations make mistakes and which rule is violated. This is helpful in fixing bugs of SSL/TLS implementations.
(2) RFCcert reveals the invalidity of certain discrepancies of traditional differential testing. Moreover, when a single implementation is tested, traditional differential testing does not work but RFCcert still works. Thus, RFCcert significantly improves traditional differential testing in a sense.
(3) The supporting tool, namely RFCcertDT, has been implemented and 89 certificates have been generated. With these certificates, 29 bugs on average in each of the 14 implementations have been found. So far, 9 reported bugs have been confirmed by Google, Microsoft, GnuTLS and wolfSSL.

The remainder of this paper is organized as follows. The next section briefly introduces X.509 certificates and certificate validation. Section 3 presents the RFC-directed certificate generation process. Subsequently, RFC-directed differential testing of certificate validation in SSL/TLS implementations is conducted in Section 4. Finally, related work is discussed in Section 5 and the paper is concluded in Section 6.



(a) General structure **(b) Extensions**

Figure 1: Structure of an X.509 certificate

2 PRELIMINARIES

This section briefly introduces the structure of an X.509 certificate and how certificates are validated.

2.1 X.509 Certificates

An X.509 certificate provides the certification of an entity's identity and public key. The certification information is stored in a complex structure which is mainly defined in RFC 5280 [15].

As Figure 1 (a) shows, an X.509 certificate consists of three parts: (1) *tbsCertificate*, (2) *signature algorithm*, and (3) *signature value*. The first part *tbsCertificate* contains the following components: (a) *version*, e.g. *v3*; (b) *serial number*, e.g. 6; (c) *signature* including identifier of the declared signature algorithm, e.g. *SHA256WithRSAEncryption*; (d) *validity* specified by a *notBefore* requirement and a *notAfter* requirement; (e) *issuer* which refers to the Certification Authority (CA) that issues the certificate; (f) *issuer unique identifier*; (g) *subject* which refers to a holder of the certificate; (h) *subject unique identifier*; (i) *subject public key info*; and (j) *extensions*. The second part *signature algorithm* presents the identifier of the signature algorithm that is actually used, e.g. *SHA1WithRSAEncryption*. As the third part, the *signature value* is recorded if *tbsCertificate* is signed by the *signature algorithm*.

As Figure 1 (b) shows, *extensions* of *tbsCertificate* contains 15 *standard extensions* and 2 *private Internet extensions*. These extensions mainly specify settings of keys, alternative names, policies, constraints, the Certificate Revocation List (CRL) and information access. The *extensions* component has an intricate relationship with *tbsCertificate*. More details about the structure can be found in RFC 5280 [15].

2.2 Certificate Validation

Certificate validation is the key of authentication since it checks the genuineness and validity of certificates.

To determine whether the certification information in an X.509 certificate is falsified, *signature value* is validated by a *signature*

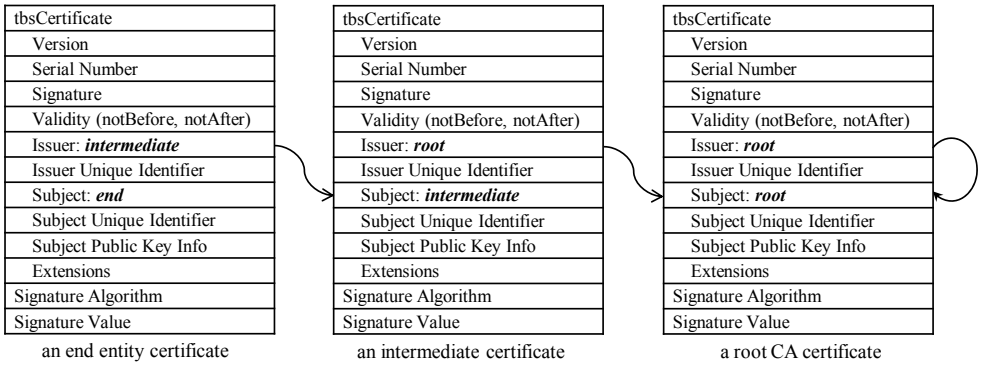


Figure 2: A certificate chain

algorithm such as *DSA* or *SHA1WithRSAEncryption*. For a trustworthy certificate, the validation checks the concrete constraints such as *notBefore*s and *notAfter*s. Besides, the validation also checks whether a certificate is issued by a valid root CA through inspecting a certificate chain. Figure 2 shows a simple certificate chain. The validation checks whether each issuer in the chain is a valid CA and whether the chain ends at a valid root CA. If all these checks pass, the current certificate is accepted, otherwise it is rejected. More information about certificate validation can be found in RFC 5246 [18] and RFC 6125 [56].

3 RFC-DIRECTED CERTIFICATE GENERATION

As Figure 3 shows, our approach RFCcert consists of four steps. At the first step, rules are extracted from RFCs and then expressed in a symbolic C program. At the second step, Dynamic Symbolic Execution (DSE) technique is employed, which produces low-level test cases. At the third step, the low-level test cases are utilized to assemble high-level test cases, i.e. certificates. Finally, the assembled certificates are used to test certificate validation in SSL/TLS implementations. This section presents the first three steps which constitute the RFC-directed certificate generation. The fourth step, i.e. differential testing of SSL/TLS implementations with the generated certificates, will be presented in Section 4.

3.1 Extracting and Expressing Rules

To generate certificates, rules are automatically extracted from RFCs 5280 [15] and 6818 [65] according to the modal key words specified by RFC 2119 [6].

3.1.1 Rules in RFCs. In RFCs 5280 and 6818, rules are expressed in two ways. Specifically, most rules are written in Natural Language (NL), while the rest are written in Abstract Syntax Notation One (ASN.1). For instance, Example 1 shows a rule in NL and Example 2 illustrates a rule in ASN.1.

Example 1. (NL) *"Conforming CAs MUST NOT issue certificates where 'policy constraints' is an empty sequence. That is, either the 'inhibitPolicyMapping' field or the 'requireExplicitPolicy' field MUST be present."*

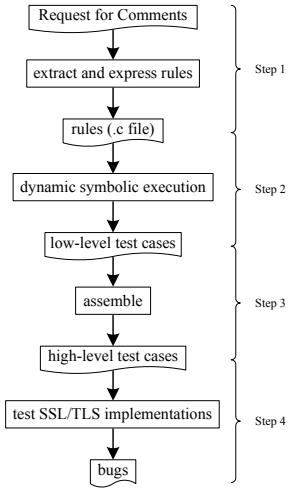


Figure 3: Overview of RFCcert

Example 2. (ASN.1) *"PolicyConstraints ::= SEQUENCE{*
requireExplicitPolicy [0] SkipCerts OPTIONAL,
inhibitPolicyMapping [1] SkipCerts OPTIONAL }"

In both NL and ASN.1, rules conform to the writing style specified by RFC 2119 [6], which remains unaltered for 20 years since its release in 1997. In particular, rules must use *modal key words* to indicate requirement levels. The modal key words include:

(1) absolute requirements or prohibitions of specifications: MUST, REQUIRED, SHALL, MUST NOT, and SHALL NOT;
(2) requirements or prohibitions with flexibility: SHOULD, RECOMMENDED, SHOULD NOT, and NOT RECOMMENDED; and
(3) truly optional items: MAY and OPTIONAL.

The usage of uppercase key words is emphasized by RFC 8174 [36] and it provides convenience for extracting rules from RFCs.

3.1.2 Extracting Rules from RFCs. A sentence or a structure is a rule if and only if it includes at least one modal key word. Algorithm 1 presents the method for extracting rules from RFC 5280 automatically. In Line 1, the mark ".\n" means that a sentence ends at the end of a line, and the mark "␣" indicates a sentence

ends in the middle of a line. In Line 2, the three marks of ASN.1 represent defined data structures without bracket, in the same line, and in different lines, respectively. In Lines 5-22, rules are extracted and saved by sections, which provides convenience for updating rules. Line 9 requires that the content of a title should not include the content of its sub-titles. In Line 16, the title nearest to a rule is used to indicate what a pronoun, e.g. "this field", points to.

Algorithm 1: Extracting rules from RFC 5280

Input: rfc5280.txt, $\min_s$=4, $\max_s$=8, $\min_t$=1, $\max_t$=5
Output: text files of rules

1 mark_{NL} = { ".\n", "␣" };
2 $\text{mark}_{ASN.1}$ = { "::=", "::={ln...}$_{ln}$", "::={ln...}$_{ln'}$" };
3 modalKeywordSet = {"MUST", ... , "OPTIONAL"};
4 R = the input txt file−{page break, header, footer, blank line};
5 R is split into sections by Level 1 titles;
6 **for** *the ith section* ($i \in \mathbb{N} \wedge min_s \leq i \leq max_s$) **do**
7 *ruleSet* = ∅;
8 *ruleNum* = 0;
9 **foreach** *content of Level j title* ($min_t \leq j \leq max_t$) **do**
10 NL sentences are extracted by mark_{NL};
11 ASN.1 structures are extracted by $\text{mark}_{ASN.1}$;
12 **foreach** $s \in$ (*NL sentence* ∪ *ASN.1 structure*) **do**
13 *wordSet*(s) = {each word of s};
14 **if** *wordSet*(s) ∩ *modalKeywordSet* ≠ ∅ **then**
15 *ruleNum*+ = 1;
16 s' = (*ruleNum*, Level j title, s);
17 *ruleSet*+ = {s'};
18 **end**
19 **end**
20 **end**
21 Save *ruleSet* as a text file named by the section number i;
22 **end**

Algorithm 1 can also be applied in automatic rule extraction from RFC 6818 by changing the input as "rfc6818.txt, $\min_s$=2, $\max_s$=8, $\min_t$=1, $\max_t$=2".

Table 1 shows the number of rules automatically extracted from each section of RFCs 5280 and 6818.

Table 1: The number of extracted rules

RFC	Sec.2	Sec.3	Sec.4	Sec.5	Sec.6	Sec.7	Sec.8
5280	-	-	227	98	19	26	13
6818	0	8	0	6	1	0	0

3.1.3 Updating Rules. As the update of RFC 5280, RFC 6818 cites the obsolete paragraphs and presents the updated paragraphs. In order to adapt to this change, Algorithm 2 is formalized for automatically updating rules. It will update Sections 4, 7 and 8 of RFC 5280 with Sections 3, 5 and 6 of RFC 6818, respectively. In Line 1, the marks "...says:", "...replaced with:", and "add(ed)..." indicate obsolete, updated, and additional paragraphs, respectively. In Line 3, obsolete rules are distinguished from updated rules according to their positions relative to the marks. In Lines 4-10, the difference between word sets of each pair of rules, is calculated and presented in a matrix. In Line 12, the updated rule, if any, which has the minimum difference from an obsolete rule is identified. In Lines 13-14, the obsolete rule is discarded if no updated rule exists. In Lines 15-18, if the obsolete and updated rules are same, the obsolete rule is kept without any changes; otherwise, it is replaced with the updated rule.

Algorithm 2: Updating rules

Input: Sections 3, 5 and 6 of RFC 6818, rules of RFC 5280
Output: updated rules of RFC 5280

1 mark_{pos} = {"...says:", "...replaced with:", "add(ed)...:" };
2 **foreach** *section of the input* **do**
3 Classify rules into *obRuleSet* and *upRuleSet* by mark_{pos};
4 **foreach** $r_1 \in$ (*obRuleSet* ∪ *upRuleSet*) **do**
5 $wordSet_{r_1}$ = {each word of r_1};
6 **foreach** $r_2 \in$ (*obRuleSet* ∪ *upRuleSet*) **do**
7 $wordSet_{r_2}$ = {each word of r_2};
8 $Matrix[r_1, r_2] = |wordSet_{r_1} - wordSet_{r_2}|$;
9 **end**
10 **end**
11 **foreach** $o \in$ *obRuleSet* **do**
12 $Update_o = u$ s.t. $\forall u' \in$ *upRuleSet*
 $Matrix[o, u] = \min(\{Matrix[o, u']\})$;
13 **if** $\nexists$ *Matrix[o,u]* **then**
14 $Update_o$ = discarded;
15 **else if** *Matrix[o,u] == Matrix[u,o]* **then**
16 $Update_o$ = unchanged;
17 **else**
18 $Update_o = u$;
19 **end**
20 Update o of RFC 5280 according to $Update_o$;
21 **end**
22 **end**

Table 2 shows the numbers of the updated and discarded rules in RFC 5280.

Table 2: Update in RFC 5280

Rules	Sec.4	Sec.5	Sec.6	Sec.7	Sec.8
Updated	3	0	0	1	0
Discarded	0	0	0	0	1

3.1.4 Consumer and Producer Rules. Rules that CAs should follow in issuing certificates are called *producer rules*, while rules required to be obeyed by certificate users during the validation process are *consumer rules*. Note that there exist rules which are both producer and consumer rules. These rules are called *shared rules*. Certificate validation in SSL/TLS implementations should conform to all the consumer-related rules. Thus, in our work, consumer

and shared rules are considered. Example 3 shows a consumer rule while Example 4 presents a shared rule.

Example 3. *"In addition, applications conforming to this profile SHOULD recognize the 'authority' and 'subject key identifier' and 'policy mappings' extensions."*

Example 4. *"This field MUST contain the same algorithm identifier as the 'signature' field in the sequence 'tbsCertificate'."*

Algorithm 3 classifies updated rules into producer, consumer and shared rules. In this algorithm, Lines 2 and 3 show explicit patterns of producer and consumer rules, respectively. Lines 5-10 add a rule to the category of producer rules if it matches the corresponding pattern. Similarly, Lines 11-16 recognize a consumer rule. Rules matching both patterns are recorded as shared rules as shown in Line 18. To prevent duplication, shared rules are removed from the categories of consumer and producer rules as shown in Lines 19 and 20, respectively. Line 21 collects implicitly shared rules such as the one in Example 4, as well as explicitly shared rules. The classification result of all the rules is shown in Table 3. Note that the sum in each column is the difference between the number in the first row of Table 1 and the number in the second row of Table 2.

Algorithm 3: Classifying rules into three categories

```
   Input: a set of updated rules (updatedRuleSet)
   Output: producer, consumer, and shared rules
1  producerRuleSet=consumerRuleSet=sharedRuleSet=∅;
2  mark_pr = {"conforming CAs", ... ,"issuing CA"};
3  mark_cr = {"implementations", ... ,"relying parties"};
4  foreach r ∈ updatedRuleSet do
5      foreach m ∈ mark_pr do
6          if r matches m then
7              producerRuleSet+ = {r};
8              break;
9          end
10     end
11     foreach m ∈ mark_cr do
12         if r matches m then
13             consumerRuleSet+ = {r};
14             break;
15         end
16     end
17 end
18 sharedRuleSet = producerRuleSet ∩ consumerRuleSet;
19 consumerRuleSet− = sharedRuleSet;
20 producerRuleSet− = sharedRuleSet;
21 sharedRuleSet =
     updatedRuleSet − producerRuleSet − consumerRuleSet;
```

3.1.5 Breakable and Unbreakable Rules. Consumer and shared rules are further classified into two categories: breakable and unbreakable rules. As an example, Example 5 indicates that the object "*explicitText* with more than 200 characters" is used to test certificate validation and we only observe the response from SSL/TLS implementations. Such a rule is unbreakable. In contrast, the rule shown in Example 4 is breakable. To violate the rule, different *signature* and *signature algorithm* could be set in a certificate, i.e., the requirement could be broken. With this setting, whether SSL/TLS implementations conform to this rule is tested. Also, bugs which violate the rule can be found in this way.

Table 3: Classification result

Rules	Sec.4	Sec.5	Sec.6	Sec.7	Sec.8
Producer	78	32	0	0	9
Consumer	40	16	12	19	2
Shared	109	50	7	7	1

Example 5. *"Therefore, certificate users SHOULD gracefully handle 'explicitText' with more than 200 characters."*

Algorithm 4 is used for classifying consumer and shared rules into breakable and unbreakable ones. The classification result is given in Table 4.

Algorithm 4: Classifying consumer and shared rules

```
   Input: consumer and shared rules (csrSet)
   Output: breakable and unbreakable rule sets
1  breakableRuleSet=unbreakableRuleSet=patternSet=∅;
2  modalKeywordSet = {"MUST", ... , "OPTIONAL"};
3  mark_ub = {"gracefully handle", ... , "receive"};
4  foreach m ∈ modalKeywordSet do
5      foreach u ∈ mark_ub do
6          patternSet+ = {m+"␣"+u};
7      end
8  end
9  foreach r ∈ csrSet do
10     foreach p ∈ patternSet do
11         if r matches p then
12             unbreakableRuleSet+ = {r};
13             break;
14         end
15     end
16 end
17 breakableRuleSet = csrSet − unbreakableRuleSet;
```

Table 4: The number of breakable/unbreakable rules

Rules	Sec.4	Sec.5	Sec.6	Sec.7	Sec.8
Breakable	90	46	5	7	1
Unbreakable	59	20	14	19	2

3.1.6 Expressing Rules as Variables. Until now, the breakable/unbreakable rules are expressed in NL/ASN.1. On the one hand, it is difficult to directly generate certificates automatically from these

rules. On the other hand, it is error prone to manually obtain combined rules. To improve this situation, RFCcert conducts DSE at the second step to automate the certificate generation process. To do so, rules are expressed as variables.

Generally, breakable and unbreakable rules present two kinds of requirements: (1) constraints on what values should be set to components of certificates, and (2) specifications on how implementations should respond to the settings. Constraints are imposed on conditions and results of a rule. In Example 4, the conditional constraint is "*tbsCertificate.signature* presents" while the resulting constraint is "*signatureAlgorithm* must contain the same *algorithm identifier*". The rule in Example 3 specifies that applications should recognize the three identifiers under the conditional constraint presented in these identifiers.

According to the constraints and specifications, these rules can be transformed into variables. The transformation is fulfilled by Algorithm 5 which consists of three steps. The first step, as shown in Lines 2-12, gathers the reserved words which constitute the backbone of rules. Then, the second step utilizes the reserved words to simplify rules as presented in Lines 13-16. After that, the third step generates combined constraints for NL and ASN.1 rules as shown in Lines 19-29 and 31-41, respectively. The number of cases attained in Line 19 is determined by the syntax of a rule. In Line 32, the embedded constraint t is usually indicated by "- - If present, . . ." in the original ASN.1 rules.

According to the algorithm, Examples 3, 4 and 5 are expressed as variables "`authorityKeyIdentifier_present_subjectKeyIdentifier_present_policyMappings_present__applications_SHOULDrecognize`", "`tbsCertSignature_present__signatureAlgorithm_MUSTsame`", and "`explicitText_Morethan200chars__certUsers_SHOULDhandle`", respectively. Due to the difficulty in natural language processing, human assistance is required to fulfill Algorithms 3-5.

3.2 Generating Low-level Test Cases

DSE fulfills on-demand combinations of rules by exploring paths of a rule-related program which is designed in agreement with testing goals. As the basis of assembling certificates, the combinations of rules are low-level test cases. With rules expressed as variables, it is possible to produce low-level test cases through the DSE technique. Specifically, we make use of the DSE tool KLEE [9] to explore paths of a symbolic program and generate low-level test cases. Algorithm 6 is formalized to automatically generate a symbolic C program from a given set of rule variables.

The structure of symbolic program that Algorithm 6 generates is shown in Figure 4. The embedded `if-else` statements avoid an exponential explosion so that KLEE will generate only $m+1$ low-level test cases, of which m are meaningful. Among the meaningful low-level test cases, i low-level test cases conform to the i unbreakable rules (Lines 8-13), and the other $m-i$ low-level test cases violate the $m-i$ breakable rules (Lines 15-20). For brevity, suppose x_1, x_2 and x_3 represent variables of Examples 3, 5 and 4, and 1 and -1 represent the bounds 2147483647 and -2147483648. Four low-level test cases will be generated by KLEE: (1) $x_1 = 1$, $x_2 = 0$ and $x_3 = 0$; (2) $x_1 = 0$, $x_2 = 1$ and $x_3 = 0$; (3) $x_1 = 0$, $x_2 = 0$ and $x_3 = -1$; and (4) $x_1 = 0$, $x_2 = 0$ and $x_3 = 0$. Due to the assignment mechanism of KLEE and the definition of (un)breakable rules, only the values 1 and -1 are meaningful. The first two low-level test cases conform to the unbreakable rules in Examples 3 and 5, respectively. The third low-level test case violates the breakable rule in Example 4.

Algorithm 5: Expressing rules as variables

```
   Input: (un)breakable rules i.e. ubrSet and brSet
   Output: variables
1  responseSet = otherSet = varSet = ∅;
2  modalKeywordSet = {"MUST", ... , "OPTIONAL"};
3  sentencePatterns = {"if", ... , "when"};
4  logicPatterns = {"and", ... , "or"};
5  responsePatterns = {"also", ... , "be prepared to"};
6  otherPatterns = {"⟨...⟩", ... , digits};
7  componentSet={components and their variants};
8  foreach r ∈ (ubrSet ∪ brSet) do
9      responseSet+ ={word after m+"␣"+rp, ∀m ∈
         modalKeywordSet and rp ∈ responsePatterns};
10     otherSet+ ={word matching p ∈ otherPatterns};
11 end
12 reservedSet = modalKeywordSet+sentencePatterns+logicPatterns
     +componentSet+responseSet+otherSet;
13 Replace pronouns with corresponding titles.
14 foreach r ∈ (ubrSet ∪ brSet) do
15     Delete w from r, ∀w ∉ reservedSet;
16 end
17 foreach r ∈ (ubrSet ∪ brSet)′ do
18     if r is a rule expressed in NL then
19         Obtain cases according to syntactic patterns;
20         foreach case of r do
21             condition="c_1, v_1, ..., c_i, v_i", i ∈ ℕ;
22             if r ∈ ubrSet then
23                 response = "consumer, actions" if any;
24                 varSet+ = condition[+" :"+response];
25             else
26                 result = "c_1, v_1, ..., c_j, v_j", j ∈ ℕ if any;
27                 varSet+ = condition[+" :"+result];
28             end
29         end
30     else
31         foreach OPTIONAL component c ∈ r do
32             if a constraint t is embedded then
33                 constraints_c = {"c, absent", t};
34             else
35                 constraints_c = {"c, absent", "c, present"};
36             end
37         end
38         foreach non-OPTIONAL component c ∈ r do
39             constraints_c = {"c, validValue"};
40         end
41         varSet+ = combined constraints for r;
42     end
43 end
44 For varSet, replace ","/":" with "_"/"__", respectively;
```

Algorithm 6: Generating a symbolic C program

Input: a set of rule variables (varSet)
Output: a symbolic C program
1 Generate "#include ..." and "int main(){";
2 Generate the variable declaration using varSet;
3 Partition varSet into ubVarSet and bVarSet by (un)breakable rules;
4 Make v symbolic, $\forall v \in$ ubVarSet;
5 Make v symbolic, $\forall v \in$ bVarSet;
6 **foreach** $v \in$ *ubVarSet* **do**
7 | Generate "if (v>0) printf(v>0) else";
8 **end**
9 **foreach** $v \in$ *bVarSet* **do**
10 | Generate "if (v<0) printf(v<0) else";
11 **end**
12 Delete the redundant "else" and add "}";
13 Save as a .c file;

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  int main(){
4      int x1, x2, ... , xm;
5      klee_make_symbolic(&x1, sizeof(x1), "x1");
6      ...
7      klee_make_symbolic(&xm, sizeof(xm), "xm");
8      if (x1>0)     //unbreakable rule
9          printf(x1>0);
10     else
11     ...
12       if (xi>0)   //unbreakable rule
13           printf(xi>0);
14       else
15         if (xj<0) //breakable rule, j=i+1
16             printf(xj<0);
17         else
18         ...
19           if (xm<0) //breakable rule
20               printf(xm<0);
21 }
```

Figure 4: Structure of a symbolic program

The fourth low-level test case is meaningless and will be omitted. From the first three low-level test cases, RFCcert will assemble certificates at the next step. The structure as shown in Figure 4 meets the requirement that each low-level test case conforms to or violates only one rule. For complicated testing goals, the structure of symbolic programs can be designed differently to realize complicated combinations of rules. Thus, DSE technique provides flexibility for generating low-level test cases to achieve different testing goals.

3.3 Assembling High-level Test Cases

Algorithm 7 uses low-level test cases to assemble certificates, i.e. high-level test cases. Basic components and their valid values are saved in Line 1. Then, Lines 3 and 4 extract components and values from conditions and results of the low-level test cases, respectively. For example, ⟨tbsCertSignature,present⟩ and ⟨signatureAlgorithm, MUSTsame⟩ come from "`tbsCertSignature_present __signatureAlgorithm_MUSTsame`". In Lines 5-7, basic components are set to default values. If a CA certificate is being assembled, Lines 8-10 set values of corresponding components. After that, Lines 11-13 set values of components according to the conditions of low-level test cases. By contrast, Lines 14-16 set values against the requirements, which actually realizes the breaking of breakable rules. For example, *signature algorithm* is set to *SHA1WithRSA* while *signature* is set to *SHA256WithRSA*. If *signature algorithm* is not involved in the rule, it will be set the same as the *signature* as shown in Lines 17-19. Finally, the certificate is encoded and saved in Line 20.

Algorithm 7: Assembling certificates

Input: low-level test cases
Output: high-level test cases i.e. certificates
1 basicMap = $\{\langle version, defaultValue\rangle, ...\}$;
2 **foreach** *low-level test case* **do**
3 | $conditionMap = \{\langle component_i, value_i\rangle\}, i \in \mathbb{N}$;
4 | $resultMap = \{\langle component_j, value_j\rangle\}, j \in \mathbb{N}$;
5 | **foreach** $c \in$ *basicMap.componentSet* **do**
6 | | Set $c = basicMap.value(c)$;
7 | **end**
8 | **if** *a CA certificate is being assembled* **then**
9 | | Set valid *basicConstraints* and *keyUsage*;
10 | **end**
11 | **foreach** $c \in$ *conditionMap.componentSet* **do**
12 | | Set $c = conditionMap.value(c)$;
13 | **end**
14 | **foreach** $c \in$ *resultMap.componentSet* **do**
15 | | Set c against $resultMap.value(c)$;
16 | **end**
17 | **if** *signatureAlgorithm has not been set* **then**
18 | | Sign *tbsCertificate* with the *SAI* in *signature*;
19 | **end**
20 | Encode the certificate and save as a file;
21 **end**

4 RFC-DIRECTED DIFFERENTIAL TESTING

In this section, an empirical study on testing certificate validation in SSL/TLS implementations is carried out. In total, 69 consumer and shared rules are used and 89 certificates are generated by RFCcertDT. These certificates act as test cases of RFC-directed differential testing.

4.1 Setup

4.1.1 Configuration. We employ 6 popular SSL/TLS implementations and 3 Web browsers for testing. Some SSL/TLS implementations have independent version series which are still maintained by developers. Thus, the following independent versions are tested: OpenSSL (v1.0.1u, v1.0.2j and v1.1.0b) [21], ARM mbedTLS v2.3.0

[37], GnuTLS (v3.3.25, v3.4.16 and v3.5.5) [44], Mozilla NSS v3.27 [48], wolfSSL v3.9.10 [63], matrixSSL v3.8.6 [57], Mozilla Firefox v49.0.1 [47], Microsoft Internet Explorer v11.0.9600.18449IS [45] and Google Chrome v54.0.2840.59 [28]. Note that mbedTLS and wolfSSL are descendants of PolarSSL and CyaSSL, respectively. For brevity, the version number of an implementation is omitted in the rest of the paper, except as otherwise noted.

Experiments are conducted on two kinds of virtual machines, both of which are configured with one core of an Intel Core i7-4790 CPU (3.60GHz) and 1.5GB RAM. One's Operating System (OS) is Canonical Ubuntu x64 v16.04-LTS and the other's is Microsoft Windows 7 x64 with service pack 1 (v6.1.7601). OpenSSL, mbedTLS, GnuTLS, NSS, wolfSSL, matrixSSL and Chrome are tested in Ubuntu. Internet Explorer is tested in Windows 7. Mozilla Firefox is tested in both of the two OSs. For accuracy, each version of SSL/TLS implementations and Web browsers is tested in a distinct virtual machine.

4.1.2 Testing Modes. SSL/TLS implementations and Web browsers provide 3 different modes for certificate validation: command-line, certificate manager, and client/server. In our experiments, each mode is adopted to perform certificate validation of a few implementations and/or browsers, in the following way.

(1) Command-line: OpenSSL, mbedTLS, GnuTLS, NSS, and matrixSSL.
(2) Certificate manager: Firefox, Internet Explorer, and Chrome.
(3) Client/server: wolfSSL.

4.1.3 Metrics. Certificates that are directly used to test certificate validation are chief certificates and certificates that are needed by chief certificates in the validation are auxiliary certificates. To analyze and evaluate testing results conveniently, we define an $(m+3)$-dimension vector $\overrightarrow{RFCcert}$ for each chief certificate:

$$\overrightarrow{RFCcert} = \langle SN, RFCreason, RFCresult, ResultI_1, ..., ResultI_m \rangle.$$

Here, *SN* is the serial number of a certificate, and *RFCreason* records a rule which is violated or obeyed. If the rule is violated, the value of *RFCresult* is 0 ("Reject"); while if the rule is obeyed, *RFCresult* is 1 ("Accept"). It is possible that both "Accept" and "Reject" are reasonable, and in this case the value of *RFCresult* is 2. Besides, $ResultI_i$ $(1 \le i \le m)$ indicates the certificate validation result of the i^{th} SSL/TLS implementation.

These vectors are used as metrics in the following ways.

(1) If $ResultI_i \neq ResultI_j (1 \le i, j \le m \wedge i \neq j)$, a discrepancy is found.
(2) For a vector $\langle ResultI_1, \ldots, ResultI_m \rangle_x$, if it is not equivalent to any $\langle ResultI_1, \ldots, ResultI_m \rangle_y$ among k discrepancies $(1 \le x, y \le k \wedge x \neq y)$, a distinct discrepancy is found.
(3) If *RFCresult* = 2, the discrepancy, if any, is invalid.
(4) *RFCreason* presents an explanation for a discrepancy.
(5) If $ResultI_i$ $(1 \le i \le m)$ does not match *RFCresult*, one bug is found and *RFCreason* provides the reason.

The total number of bugs found in the i^{th} SSL/TLS implementation is calculated by the following formula.

$$N_b(i) = \sum_{\substack{\overrightarrow{RFCcert} \\ s.t.\ RFCresult \neq 2}} |RFCresult - ResultI_i| \quad (1)$$

We define the conformance ratio as a criterion for sorting SSL/TLS implementations:

$$CR(i) = \frac{|RFCresult \text{ matches } ResultI_i|}{|certs|} \times 100\% \quad (2)$$

where $|RFCresult$ matches $ResultI_i|$ is the number of experimental results in accordance with RFCs and $|certs|$ is the number of chief certificates.

4.2 Comparing Mucert with RFCcert

Frankencert [8] is the pioneering work in automated testing of certificate validation in SSL/TLS implementations. As mentioned previously, experiments in [11] show that Mucert is more efficient than Frankencert in finding distinct discrepancies. Therefore, we compare RFCcert with Mucert.

Table 5 and Figure 7 of [11] show that *mucert-3* with parameters $\beta = -0.3$ and $k = 500$ finds more distinct discrepancies than others. Thus, the parameters are employed in the repetition of Mucert and the mutated certificates are employed in the comparison. There are two schemes for comparison. One scheme is to test old SSL/TLS implementations adopted in [11]. However, neither obsolete versions, e.g. matrixSSL v3.7.1, nor SHA256/SHA1/MD5 values are publicly available, and experiments on non-official versions are not credible. The other scheme is to test new versions as listed in Section 4.1.1. For this scheme, there exist two problems. First, new versions of Firefox and Chrome require certificates in the format of *.p12/pfx*, but certificates of Mucert cannot be converted to this format since they do not match the private key. Second, the CA certificate of Mucert cannot be loaded without modification of wolfSSL's source code. For the sake of fairness and usefulness in practice, Mucert and RFCcert are compared on OpenSSL, mbedTLS, GnuTLS, NSS, matrixSSL and Internet Explorer (10 versions in total). We have conducted experiments with 89 certificates on these 10 implementations. The results show that RFCcert finds 12 distinct discrepancies while Mucert finds 8 with 2 in common. Therefore, RFCcert is a significant supplement to Mucert in fining distinct discrepancies. Table 5 shows the comparison in detail.

Table 5: A comparison of RFCcert with Mucert

Item	Mucert	RFCcert
Distinct discrepancies	8	12
Number of Certificates (NC)	1005	89
Time for certificate generation	10445	15
NC for one distinct discrepancy	125.63	7.42

The following facts can be drawn from Table 5.

(a) RFCcert has found 12 distinct discrepancies by using just 89 certificates while Mucert has found 8 distinct discrepancies by using 1,005 certificates. Therefore, for Finding Ratio (FR), RFCcert is 12/89=13.48% while Mucert is 8/1005=0.796%. It is clear that RFCcert is more effective than Mucert.
(b) RFCcert spends 15 seconds in generating 89 certificates while Mucert spends 10,445 seconds for generating 1,005 certificates. For the average Speed of Finding (SF) a distinct discrepancy,

RFCcert is 12/15=0.8 while Mucert is 8/10445=0.000766. Therefore, RFCcert is more efficient than Mucert.

4.3 Invalidity of Traditional Differential Testing of Certificate Validation

Neither Frankencert nor Mucert shows directly whether a discrepancy is valid. By contrast, RFCcert can determine whether a discrepancy is valid. In fact, if the value of *RFCresult* is 2, the discrepancy is invalid. Example 6 presents a rule that "this extension", i.e. *basic constraints*, may be *critical* or *non-critical*. One certificate of RFCcert is assembled for this rule and *basic constraints* is set to *critical*. Our experiments find one discrepancy related to the certificate: wolfSSL and matrixSSL reject it while other implementations accept it. Consequently, non-restricted "MAY/OPTIONAL" rules indicate that differential testing of certificate validation in SSL/TLS implementations may be invalid in some cases.

Example 6. *"This extension MAY appear as a critical or non-critical extension in end entity certificates."*

In addition, traditional differential testing requires at least two SSL/TLS implementations. Otherwise it cannot find any discrepancy. In contrast, RFCcert is able to test a single implementation by checking whether $RFCresult_x$ matches $ResultI_{i,x}$ for some i ($1 \leq i \leq m$) and for all x ($1 \leq x \leq |certs|$). If the match fails, $RFCreason_x$ indicates which rule is violated in the implementation.

4.4 Finding and Reporting Bugs

4.4.1 Finding Bugs and Conformance Ratios. Fourteen versions of implementations mentioned in Section 4.1.1 are tested individually by the 89 certificates produced by RFCcert and the numbers of the detected bugs are shown in Figure 5. It shows that RFCcert finds 29 bugs on average in each of the 14 implementations.

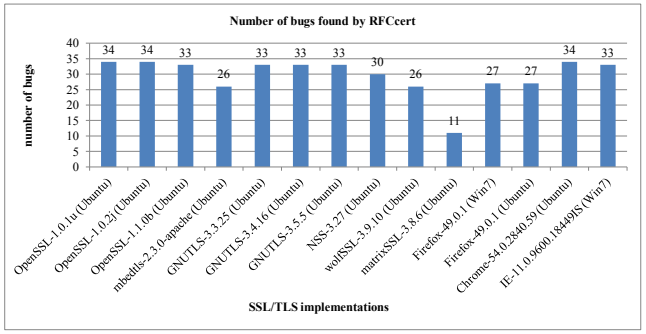


Figure 5: Number of bugs found by RFCcert

Based on current test cases and Formula (2) where the number of chief certificates is 79, SSL/TLS implementations are sorted by the conformance ratios in Figure 6. The ranking is: (1) matrixSSL, (2) wolfSSL and mbedTLS, (3) Firefox (in Windows and Ubuntu), (4) NSS, (5) OpenSSL v1.1.0b, GnuTLS and IE, and (6) Chrome, OpenSSL v1.0.1u and 1.0.2j. The ranking provided by RFCcert is helpful for users to choose appropriate SSL/TLS implementations in terms of adherence to RFCs.

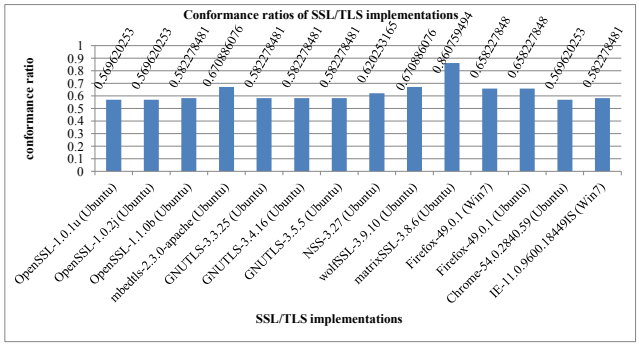


Figure 6: Conformance ratios

4.4.2 Reporting Bugs and Feedback. We have been reporting bugs found by RFCcert to the corresponding developers. Up to now, at least 9 bugs have been confirmed or fixed.

Google has confirmed and fixed one reported bug. We have reported the security bug to Google that Chrome in Ubuntu accepts a certificate whose *tbsCertificate.signature* and *Certificate.signatureAlgorithm* are different. If a strong hash algorithm, e.g. *SHA256*, is declared in *tbsCertificate.signature* while a weak hash algorithm, e.g. *SHA1*, is actually used in *Certificate.signatureAlgorithm*, policy checks prohibiting weak signature will pass because they only care about *tbsCertificate.signature* in the certificate. If an attacker obtains such a certificate which has a weak signature algorithm, he may crack it and generate a forged certificate at his will. The forged certificate will be falsely accepted by clients/servers, which may result in security problems. Google Security Team has carried out experiments on eight permutations of *Certificate.signatureAlgorithm*, *Certificate.signatureValue* and *tbsCertificate.signature* and confirmed that it is a security bug. The team has fixed this security bug in Chrome v58.0.3029.81 (CVE-2017-5066)[1,2].

Microsoft has confirmed one reported bug. We have reported the similar security bug found in Microsoft Internet Explorer. Microsoft Edge Team has confirmed and appreciated our report[3].

Although it has been fixed in the code of verifier of Chrome v58.0.3029.81, this security bug still exists in the underlying platform verifier in Ubuntu and Windows. That is, other untested Web browsers and SSL/TLS implementations depending on the OS verifier may have similar security bugs and their users are vulnerable to attacks which exploit this bug.

GnuTLS has confirmed and fixed three bugs: (1) GnuTLS accepts a version-1 certificate with *issuer unique identifier*[4]; (2) GnuTLS accepts a version-1 certificate with *subject unique identifier*[5]; and (3) GnuTLS accepts a certificate which has a fractional validity[6].

wolfSSL has confirmed and fixed four bugs[3]: (1) wolfSSL accepts a certificate with duplicate *policy object identifiers*; (2) wolfSSL accepts a certificate with more than one instance of a particular

[1]https://chromereleases.googleblog.com/2017/04/stable-channel-update-for-desktop.html
[2]https://nvd.nist.gov/vuln/detail/CVE-2017-5066
[3]Reports are not public and screen shots are provided upon request.
[4]https://gitlab.com/gnutls/gnutls/issues/167
[5]https://gitlab.com/gnutls/gnutls/issues/168
[6]https://gitlab.com/gnutls/gnutls/issues/169

extension; (3) wolfSSL accepts a certificate with a relative URI in *subject alternative name*; and (4) wolfSSL accepts an end entity certificate with *name constraints*.

To sum up, RFCcert is more cost-efficient than the state-of-the-art approach Mucert [11], and traditional differential testing of certificate validation in SSL/TLS implementations is sometimes invalid. More importantly, RFCcert is capable of identifying bugs in a single SSL/TLS implementation and providing RFC-based explanations, while traditional differential testing cannot fulfill these tasks.

5 RELATED WORK

Several vulnerabilities in certificate validation of SSL/TLS implementations have been reported in [39–42]. Kaminsky et al. show a case of MITM attacks by exploiting a flaw in certificate validation [32]. Georgiev et al. study bugs of APIs of SSL/TLS implementations and analyze perils and pitfalls of certificate validation based on these APIs [23]. These work shows that certificate validation in SSL/TLS implementations is insecure and it is necessary to test certificate validation.

Due to the intricate syntactic and semantic constraints of certificates, methods that have succeeded in other fields are difficult to be employed in certificate validation. In [17], Daniel et al. generate structurally complex Java programs iteratively and employ them to test refactoring engines. However, this method is not suitable for testing certificate validation due to the illegal structures for most of the test cases. Fuzzing [26, 27, 29, 61] provides few useful test cases for certificate validation either, since most test cases do not have the complex structures required by certificates. Besides, genetic algorithms have been adopted to test certificate validation. Frankencert [8] is the first automated differential testing of certificate validation and it produces certificates by mutation. It finds 9 distinct discrepancies [11] by using 8,127,600 randomly mutated certificates. The aimless mutation leads to Frankencert's low efficiency in finding discrepancies. Inspired by the idea of feedback-directed test case generation [50–52], Mucert considers code coverage to guide certificate mutation. Thus the mutated certificates are related to the code of certificate validation in OpenSSL. Mucert also employs MCMC sampling to diversify the mutated certificates. In [11], Mucert finds 27 distinct discrepancies by using 1,005 mutated certificates. Therefore, Mucert significantly improves efficiency of differential testing of certificate validation in SSL/TLS implementations.

However, there are limitations in genetic-algorithm-based approaches. The choice of seed certificates and mutation operators has a great effect on the performance of Frankencert and Mucert, and it is time-consuming to choose appropriate seed certificates and mutation operators. For a mutated certificate which leads to a discrepancy, its structure and content are not clear until they are analyzed manually. Hence, neither Frankencert nor Mucert can directly point out: whether or why a discrepancy is invalid; or which implementations make mistakes. Moreover, to solve a discrepancy, manual work is required for analyzing intricate constraints of the structure and content in a mutated certificate. In particular, for a single implementation, traditional differential testing using genetic approaches does not work. Although the state-of-the-art approach Mucert is guided by the code coverage of OpenSSL, different implementations have different codes and the code coverage of OpenSSL cannot represent that of others. It is still an open question how to obtain an overall code coverage to guide certificate mutation. Therefore, a better way is to generate discrepancy-targeted certificates directed by high-level specifications instead of the code coverage.

Specifications have been used for various purposes. Grammar-based testing techniques [33, 43, 59, 64] generate programs based on grammars. For example, Csmith [64] employs generated programs to find bugs in open-source C compilers. Korat [5] employs the precondition of a method to generate test cases and then employs the postcondition as a test oracle to check the correctness of outputs. TestEra [38] produces Java programs as counterexamples to violated correctness criteria. Hierons et al. have made a survey of using formal specification to assist testing [30].

Inspired by the above specification-based methods, RFCcert utilizes DSE [9, 12, 13, 25, 27, 58, 60] and assembling techniques [1, 10, 16] to efficiently generate standards-based certificates to overcome the disadvantages of traditional differential testing. With the standards, certificates produced by RFCcert are more discrepancy-targeted than mutated certificates generated by genetic approaches such as Frankencert and Mucert. Experimental results show that RFCcert significantly improves the differential testing of certificate validation in SSL/TLS implementations. In addition, RFCcert can be extended to specification-based differential testing of other implementations.

Suman et al. employ error specifications to detect error handling bugs [31]. RFCcert is different in that we employ standards to generate certificates with complex structures and then employ them to find bugs in certificate validation. In [3], the approach proposed by Bauer et al. is a model-based run-time verification technique which completely depends on the open-source code. By contrast, our approach is an RFC-directed testing technique where the code is not required.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose RFCcert, an RFC-directed approach to the testing certificate validation in SSL/TLS implementations. With RFC-based certificates, RFCcert reveals bugs, identifies distinct discrepancies, and determines the validity of differential testing. Extensive experiments show that RFCcert is effective in finding distinct discrepancies and bugs.

In the near future, we are going to apply our approach to all the rules in RFCs 5280 and 6818 for detecting vulnerabilities or bugs. In addition, we plan to improve our tool RFCcertDT so as to help developers enhance the security of certificate validation in SSL/TLS implementations. We look forward to facilitating the standardization of certificate validation in SSL/TLS implementations and making it convenient for users to choose suitable products according to the conformance ratios.

ACKNOWLEDGMENTS

This research was supported by NSFC with Grant Nos. 61732013, 61420106004, 61751207 and 61402347. Cong Tian and Zhenhua Duan are the corresponding authors.

REFERENCES

[1] A. Aliprandi, M. Mauro, and L. De Cola. 2016. Controlling and Imaging Biomimetic Self-assembly. *Nature Chemistry* 8, 1 (2016), 10–15.

[2] R. Barnes, M. Thomson, A. Pironti, and A. Langley. 2015. Deprecating Secure Sockets Layer Version 3.0. (June 2015). https://tools.ietf.org/html/rfc7568

[3] A. Bauer, J. Jürjens, and Y. Yu. 2011. Run-Time Security Traceability for Evolving Systems. *Comput. J.* 54, 1 (January 2011), 58–87.

[4] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. 2015. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. (September 2015). https://tools.ietf.org/html/rfc7627

[5] C. Boyapati, S. Khurshid, and D. Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 123–133. http://doi.acm.org/10.1145/566172.566191

[6] S. Bradner. 1997. Key Words for Use in RFCs to Indicate Requirement Levels. (March 1997). https://tools.ietf.org/html/rfc2119

[7] M. Brown and R. Housley. 2010. Transport Layer Security (TLS) Authorization Extensions. (May 2010). https://tools.ietf.org/html/rfc5878

[8] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 114–129. https://doi.org/10.1109/SP.2014.15

[9] C. Cadar, D. Dunbar, and D. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[10] A. Casini, M. Storch, G. S. Baldwin, and T. Ellis. 2015. Bricks and Blueprints: Methods and Standards for DNA Assembly. *Nature Reviews Molecular Cell Biology* 16, 9 (2015), 568–576.

[11] Y. Chen and Z. 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 (ESEC/FSE 2015)*. ACM, New York, NY, USA, 793–804. https://doi.org/10.1145/2786805.2786835

[12] V. Chipounov, V. Kuznetsov, and G. Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[13] V. Chipounov, V. Kuznetsov, and G. Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.* 30, 1, Article 2 (Feb. 2012), 49 pages. https://doi.org/10.1145/2110356.2110358

[14] S. Chokhani and W. Ford. 1999. Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework. (March 1999). https://tools.ietf.org/html/rfc2527

[15] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. (May 2008). https://tools.ietf.org/html/rfc5280

[16] J. R. Crandall, R. Ensafi, S. Forrest, J. Ladau, and B. Shebaro. 2008. The Ecology of Malware. In *Proceedings of the 2008 Workshop on New Security Paradigms (NSPW '08)*. ACM, New York, NY, USA, 99–106. https://doi.org/10.1145/1595676.1595692

[17] B. Daniel, D. Dig, K. Garcia, and D. Marinov. 2007. Automated Testing of Refactoring Engines. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 185–194. https://doi.org/10.1145/1287624.1287651

[18] T. Dierks and E. Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. (August 2008). https://tools.ietf.org/html/rfc5246

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. 1997. Hypertext Transfer Protocol – HTTP/1.1. (January 1997). https://tools.ietf.org/html/rfc2068

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. 1999. Hypertext Transfer Protocol – HTTP/1.1. (June 1999). https://tools.ietf.org/html/rfc2616

[21] OpenSSL Software Foundation. 2016. OpenSSL. (2016). Retrieved October 12, 2016 from https://www.openssl.org

[22] A. Freier, P. Karton, and P. Kocher. 2011. The Secure Sockets Layer (SSL) Protocol Version 3.0. (August 2011). https://tools.ietf.org/html/rfc6101

[23] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. 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 (CCS '12)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2382196.2382204

[24] D. Gillmor. 2016. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). (August 2016). https://tools.ietf.org/html/rfc7919

[25] P. Godefroid, N. Klarlund, and K. Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[26] P. Godefroid, M. Y. Levin, and D. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium*. The Internet Society, San Diego, California, USA, 151–166.

[27] P. Godefroid, M. Y. Levin, and D. Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[28] Google. 2016. Chrome. (2016). Retrieved October 12, 2016 from https://www.google.com/chrome/

[29] G. Grieco, M. Ceresa, and P. Buiras. 2016. QuickFuzz: An Automatic Random Fuzzer for Common File Formats. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 13–20. https://doi.org/10.1145/2976002.2976017

[30] R. Hierons, K. Bogdanov, J. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. Simons, S. Vilkomir, M. Woodward, and H. Zedan. 2009. Using Formal Specifications to Support Testing. *ACM Comput. Surv.* 41, 2, Article 9 (Feb. 2009), 76 pages. https://doi.org/10.1145/1459352.1459354

[31] S. Jana, Y. J. Kang, S. Roth, and B. Ray. 2016. Automatically Detecting Error Handling Bugs Using Error Specifications. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 345–362.

[32] D. Kaminsky, M. L. Patterson, and L. Sassaman. 2010. PKI Layer Cake: New Collision Attacks Against the Global X.509 Infrastructure. In *Proceedings of the 14th International Conference on Financial Cryptography and Data Security (FC'10)*. Springer-Verlag, Berlin, Heidelberg, 289–303. https://doi.org/10.1007/978-3-642-14577-3_22

[33] R. Lämmel and W. Schulte. 2006. Controllable Combinatorial Coverage in Grammar-based Testing. In *Proceedings of the 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom'06)*. Springer-Verlag, Berlin, Heidelberg, 19–38. https://doi.org/10.1007/11754008_2

[34] A. Langley. 2015. A Transport Layer Security (TLS) ClientHello Padding Extension. (October 2015). https://tools.ietf.org/html/rfc7685

[35] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson. 2016. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). (June 2016). https://tools.ietf.org/html/rfc7905

[36] B. Leiba. 2017. Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. (May 2017). https://tools.ietf.org/html/rfc8174

[37] ARM Limited. 2016. mbedTLS. (2016). Retrieved October 12, 2016 from https://tls.mbed.org

[38] D. Marinov and S. Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE '01)*. IEEE Computer Society, Washington, DC, USA, 22–31. http://dl.acm.org/citation.cfm?id=872023.872551

[39] M Marlinspike. 2002. IE SSL Vulnerability. (2002). Retrieved October 1, 2016 from https://www.thoughtcrime.org/ie-ssl-chain.txt

[40] M. Marlinspike. 2009. More Tricks for Defeating SSL in Practice. (2009). Retrieved October 1, 2016 from https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf

[41] M. Marlinspike. 2009. New Tricks for Defeating SSL in Practice. (2009). Retrieved October 1, 2016 from https://www.blackhat.com/presentations/bh-usa-09/Marlinspike/BHUSA09-Marlinspike-DefeatSSL-SLIDES.pdf

[42] M. Marlinspike. 2009. Null Prefix Attacks against SSL/TLS Certificates. (2009). Retrieved October 1, 2016 from https://www.thoughtcrime.org/papers/null-prefix-attacks.pdf

[43] P. M. Maurer. 1990. Generating Test Data with Enhanced Context-free Grammars. *IEEE Software* 7, 4 (July 1990), 50–55. https://doi.org/10.1109/52.56422

[44] N. Mavrogiannopoulos. 2016. GnuTLS. (2016). Retrieved October 12, 2016 from https://www.gnutls.org

[45] Microsoft. 2016. Internet Explorer. (2016). Retrieved October 12, 2016 from https://www.microsoft.com/en-us/download/internet-explorer.aspx

[46] B. Moeller and A. Langley. 2015. TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks. (April 2015). https://tools.ietf.org/html/rfc7507

[47] Mozilla. 2016. Firefox. (2016). Retrieved October 12, 2016 from https://www.mozilla.org/en-US/firefox/all/

[48] Mozilla. 2016. NSS. (2016). Retrieved October 12, 2016 from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/NSS_Releases

[49] NIST. 2017. CVE-2016-8495. (2017). Retrieved July 26, 2017 from https://nvd.nist.gov/vuln/detail/CVE-2016-8495

[50] C. Pacheco and M. D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA '07)*. ACM, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[51] C. Pacheco, S. K. Lahiri, and T. Ball. 2008. Finding Errors in .Net with Feedback-directed Random Testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 87–96. https://doi.org/10.1145/1390630.1390643

[52] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 75–84. https://doi.org/10.1109/ICSE.2007.37

[53] A. Popov. 2015. Prohibiting RC4 Cipher Suites. (February 2015). https://tools.ietf.org/html/rfc7465

[54] E. Rescorla. 2000. HTTP Over TLS. (May 2000). https://tools.ietf.org/html/rfc2818

[55] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. 2010. Transport Layer Security (TLS) Renegotiation Indication Extension. (February 2010). https://tools.ietf.org/html/rfc5746

[56] P. Saint-Andre and J. 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). (March 2011). https://tools.ietf.org/html/rfc6125

[57] Inside Secure. 2016. matrixSSL. (2016). Retrieved October 12, 2016 from http://www.matrixssl.org

[58] K. Sen and G. Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. Springer-Verlag, Berlin, Heidelberg, 419–423. https://doi.org/10.1007/11817963_38

[59] E. G. Sirer and B. N. Bershad. 1999. Using Production Grammars in Software Testing. In *Proceedings of the 2nd Conference on Domain-specific Languages (DSL '99)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/331960.331965

[60] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[61] M. Sutton, A. Greene, and P. Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, Boston, Massachusetts, USA.

[62] S. Turner and T. Polk. 2011. Prohibiting Secure Sockets Layer (SSL) Version 2.0. (March 2011). https://tools.ietf.org/html/rfc6176

[63] wolfSSL Inc. 2016. wolfSSL. (2016). Retrieved October 30, 2016 from https://www.wolfssl.com

[64] X. Yang, Y. Chen, E. Eide, and J. Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[65] P. Yee. 2013. Updates to the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. (January 2013). https://tools.ietf.org/html/rfc6818