Coverage-directed Differential Testing of X.509 Certificate Validation in SSL/TLS Implementations

PENGBO NIE, Shanghai Jiao Tong University, China
CHENGCHENG WAN, University of Chicago, United States
JIAYU ZHU, Shanghai Jiao Tong University, China
ZIYI LIN, Alibaba Group Inc., China
YUTING CHEN, Shanghai Jiao Tong University, China
ZHENDONG SU, ETH Zurich, Switzerland

Secure Sockets Layer (SSL) and Transport Security (TLS) are two secure protocols for creating secure connections over the Internet. X.509 certificate validation is important for security and needs to be performed before an SSL/TLS connection is established. Some advanced testing techniques, such as frankencert, have revealed, through randomly mutating Internet accessible certificates, that there exist unexpected, sometimes critical, validation differences among different SSL/TLS implementations. Despite these efforts, X.509 certificate validation still needs to be thoroughly tested as this work shows.

This article tackles this challenge by proposing transcert, a coverage-directed technique to much more effectively test real-world certificate validation code. Our core insight is to (1) leverage easily accessible Internet certificates as seed certificates and (2) use code coverage to direct certificate mutation toward generating a set of diverse certificates. The generated certificates are then used to reveal discrepancies, thus potential flaws, among different certificate validation implementations.

We implement transcert and evaluate it against frankencert, NEZHA, and RFCcert (three advanced fuzzing techniques) on five widely used SSL/TLS implementations. The evaluation results clearly show the strengths of transcert: During 10,000 iterations, transcert reveals 71 unique validation differences, 12×, 1.4×, and 7× as many as those revealed by frankencert, NEZHA, and RFCcert, respectively; it also supplements RFCcert in conformance testing of the SSL/TLS implementations against 120 validation rules, 85 of which are exclusively covered by transcert-generated certificates. We identify 17 root causes of validation differences, all of which have been confirmed and 11 have never been reported previously. The transcert-generated X.509 certificates also reveal that the primary goal of certificate chain validation is stated ambiguously in the widely adopted public key infrastructure standard RFC 5280.

CCS Concepts: • **Software and its engineering → Software testing and debugging;**

This research is supported by National Natural Science Foundation of China (Grant No. 62032004). This work was supported by Alibaba Group through Alibaba Innovative Research (AIR) programme. Yuting Chen was also supported by CCF-Huawei Innovative Research programme (TC20210701006/CCF2021-admin-270).

Authors' addresses: P. Nie, J. Zhu, and Y. Chen (corresponding author), Shanghai Jiao Tong University, Department of Computer Science and Engineering, Dongchuan Road No. 800, Shanghai, Shanghai, China, 200240; emails: {yuemonangong, joky27, chenyt}@sjtu.edu.cn; C. Wan, University of Chicago, United States; email: cwan@uchicago.edu; Z. Lin, Alibaba Shanghai R&D Center, Shanghai, China; email: cengfeng.lzy@alibaba-inc.com; Z. Su, ETH Zurich, Switzerland; email: zhendong.su@inf.ethz.ch.

1049-331X/2023/02-ART3 $15.00
https://doi.org/10.1145/3510416

Additional Key Words and Phrases: Coverage transfer graph, differential testing, certification mutation, certificate validation

ACM Reference format:
Pengbo Nie, Chengcheng Wan, Jiayu Zhu, Ziyi Lin, Yuting Chen, and Zhendong Su. 2023. Coverage-directed Differential Testing of X.509 Certificate Validation in SSL/TLS Implementations. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 3 (February 2023), 31 pages.
https://doi.org/10.1145/3510416

1 INTRODUCTION

Secure Sockets Layer (SSL) [24] and **Transport Security (TLS)** [44] are two secure protocols for establishing secure connections over the Internet. They are widely used in practice to verify subjects in communication and encrypt the data in transit.

Certificate validation, the most dangerous code in the world [25], plays a key role in establishing secure connections. It validates an X.509 certificate against an authority before an SSL/TLS connection is established. An X.509 certificate is a signed data structure that binds a public key to a person, computer, or organization. Certificate validation then follows RFC 5280—the widely accepted international X.509 **public key infrastructure (PKI)** standard [18]—to verify that a public key belongs to the user, computer, or service identity contained within the certificate.

Existing SSL/TLS implementations, including web browsers and SSL/TLS libraries, such as OpenSSL,[1] mbed TLS,[2] GnuTLS,[3] MatrixSSL,[4] and Mozilla's NSS,[5] have their X.509 certificate validation. Given an end entity certificate, these implementations should consistently check its validity and as well whether it is issued by a trusted **Certification Authority (CA)**. But similarly to any real-world software, SSL/TLS implementations or libraries may contain defects and in particular may not validate X.509 certificates correctly, making SSL/TLS connections vulnerable or insecure.

Brubaker et al. [10] propose frankencert, the first automated technique that randomly combines parts of real certificates for differentially testing various SSL/TLS implementations. This is a strong effort, but the "blind" nature of frankencert makes it cost-ineffective: An enormous number of frankencerts are generated, while most of them are redundant during testing; indeed, 8,127,600 frankencerts could only yield 208 discrepancies, which are further reduced to only 9 distinct ones, among the many SSL/TLS implementations [10]. Petsio et al. propose NEZHA [43], which randomly chooses and mutates seeding certificates and only maintains the ones if they can trigger behavioral asymmetries that have never seen before. As a result, NEZHA's certificate generation process converges; after 10,000 iterations, NEZHA can generate a new certificate during 270 iterations on average. Tian et al. propose RFCcert [50], a rule-based approach to generating certificates by deliberately meeting or violating RFC rules; much human effort needs to be spent on extracting and formalizing these rules.

A further investigation reveals that existing testing techniques are less effective in deeply testing SSL/TLS implementations, mainly due to two limitations.

Limitation 1. *The key components of certificate validation are interleaved and thus are not easy to be sufficiently tested.* As Figure 1 shows, certificate validation establishes a *certificate chain* (starting with a CA certificate or a trust anchor and ending with an end entity certificate) followed by

[1] https://www.openssl.org.
[2] https://tls.mbed.org.
[3] https://www.gnutls.org.
[4] https://www.matrixssl.org.
[5] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

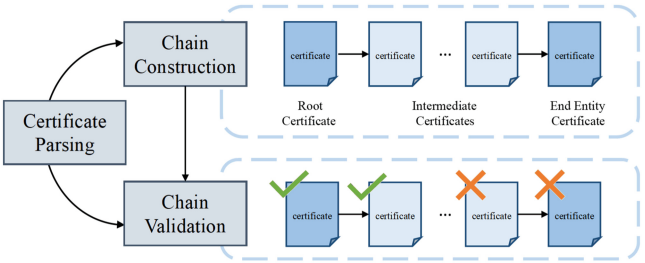


Fig. 1. A typical certificate validation process.

checking the validity of each certificate on the chain. There are three key components in certificate validation: **Certificate parsing (CP)** is responsible for parsing certificates, **chain construction (CC)** is for constructing a certificate chain, and **chain validation (CV)** is for validating the end entity certificate against a trust anchor. These three components are interleaved with each other, making it difficult to deeply test certificate validation. Most of the validation differences revealed by existing techniques are then raised, because SSL/TLS implementations are differently error prone in parsing illegal certificates. Few mutants can reveal deep differences in the other two components.

Limitation 2. *The diversity of the certificate mutants is not guaranteed.* Existing fuzz testing techniques take some random strategies to mutate certificates, expecting corner cases to appear by random. For example, frankencert randomly crossovers two or more certificates. Meanwhile, the randomness introduced into fuzz testing does not guarantee or even measure the diversity of the test suite.

Inspired by the previous techniques and recognizing their limitations, this article proposes transcert, a coverage-directed approach to deep testing of X.509 certificate validation in SSL/TLS implementations. The goal of transcert is to generate a set of diverse test certificates. By diverse, we mean, for example, that some certificates should pass validation and some should not, the certificates take different control-flow paths, and they enforce various validation policies or lead to different types of exceptions. To this end, we employ code coverage to measure the diversity of the test suite and direct certificate mutations toward generating much more diverse test certificates.

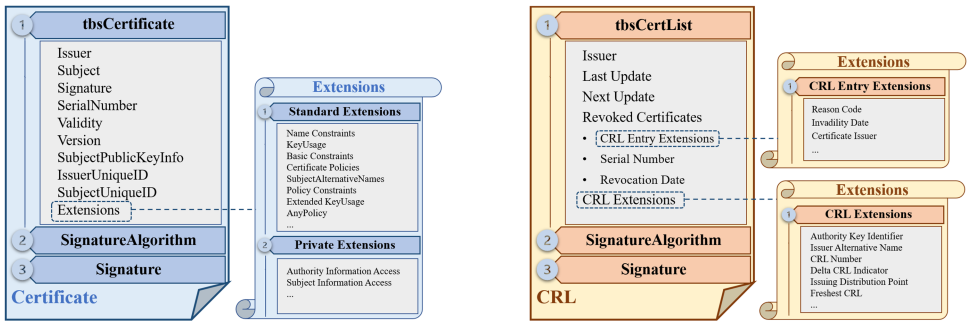
This article makes the following contributions:

Problem. We cast the difficult problem of certificate mutation as a space-exploration problem, which allows us to leverage many easily accessible Internet certificates and mutate them for effectively testing certificate validation logic. This high-level view is general and may be applicable in other settings with structurally complex test inputs.

Approach. We propose a coverage-directed approach to differential testing of certificate validation in SSL/TLS implementations. Given a set of seeding certificates, the approach approximates their diversity by running them on a reference SSL/TLS implementation followed by collecting their code coverage statistics; the approach mutates these seeds and directs certificate mutation toward generating much more diverse X.509 certificate mutants. Here a *coverage transfer graph* is designed to abstract the executions of the test certificates and the coverage transfers from the seeds to the mutants.

Evaluation and analysis. We implement transcert and evaluate it against frankencert, NEZHA,[6] and RFCcert on five widely used SSL/TLS implementations. The evaluation results clearly show the

[6]The evaluation is conducted on a re-implemented NEZHA. The original NEZHA tool depends on some out-of-date libraries and cannot be compiled successfully.



(a) Structure of an X.509 certificate. (b) Structure of a certificate revocation list.

Fig. 2. A certificate and a certificate revocation list.

strengths of transcert: During 10,000 iterations, transcert reveals 71 unique validation differences, 12× and 1.4× as many as those revealed by frankencert and NEZHA, respectively. transcert also supplements RFCcert in conformance testing of certificate validation implementations against 85 exclusive validation rules and triggering 7× more discrepancies. We analyze these validation differences and identify 17 root causes, all of which have been confirmed, and at least 11 have never been reported previously. The transcert-generated X.509 certificates also reveal that the primary goal of certificate chain validation is stated ambiguously in the widely adopted PKI standard RFC 5280.

The rest of this article is organized as follows. Section 2 introduces the background knowledge of X.509 certificate, certificate revocation list, and certificate validation. Section 3 presents the mutators designed in the study. Section 4 presents the technical details of transcert. Section 5 evaluates transcert on five SSL/TLS implementations. Section 6 analyzes root causes to validation differences revealed by transcert. Section 7 discusses related work and Section 8 concludes.

2 BACKGROUND

This section discusses necessary background on certification validation.

2.1 X.509 Certificate and Certificate Revocation List

Given an X.509 certificate, certificate validation checks whether its subject and the public key are trustworthy. An X.509 certificate, as Figure 2(a) shows, contains a sequence of three required fields: (1) a *tbscertificate*, which contains a subject and an issuer, a public key associated with the subject, a validity period, and so on (an X.509 v3 certificate also has extensions that can convey data such as additional subject identification information, policy information, and certification path constraints); (2) a *signature algorithm* used by a CA to sign this certificate; and (3) a digital *signature* for the certificate.

Certificates can be revoked; each CA will periodically publish a **certificate revocation list (CRL)**. A CRL, as Figure 2(b) shows, contains a sequence of three fields: (1) a *tbsCertList*, which contains an issuer, a validity period, an optional list of *revoked certificates*, and optional CRL extensions; (2) a *signature algorithm*; and (3) a *signature*.

2.2 Certificate Validation

A certificate validation implementation verifies an end entity certificate through establishing a certificate chain followed by checking the validity of each certificate on the chain. A certificate

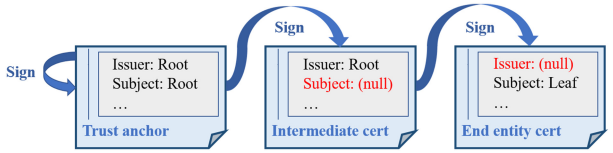


Fig. 3. A certificate chain that triggers a validation difference. The chain is mutated from a validated certificate chain by deleting the subject name of the intermediate certificate and the issue name of the end entity certificate. The root cause to the validation difference is that some certificates with empty issuer/subject names can/cannot be chained.

chain is a sequence of certificates that connects a root certificate to the end entity certificate, each one having signed the next certificate. Here the *root certificate*, which is also called *a trust anchor*, is a certificate containing a trusted CA and the public key. The other certificates on the chain are called *intermediate certificates*.

Certificate validation is composed of three components:

- **CP.** The CP component is responsible for parsing the certificates in a repository for constructing a certificate chain; it also recognizes the fields and extensions of each certificate on the certificate chain when the chain is validated.
- **CC.** Typically, a chain can be constructed by starting from the end entity certificate and successively retrieving the issuer certificates until a trust anchor is found. Chain construction is non-trivial, because the PKI structure could be complicated, as RFC 4158 [19] explains.
- **CV.** RFC 5280 describes a reference chain validation process, which starts from the trust anchor and validates each certificate on the chain successively: checking its digital signature, validity period, and so on. Constraints specified in certificate extensions, such as the name constraints, policy constraints, and so on, are also used in chain validation.

During certificate validation, SSL/TLS implementations do not only verify the signature and validity of the certificate but also obtain recent CRLs to determine whether the target certificate and/or the intermediate ones have been revoked. This process is also called *CRL validation*.

2.3 Validation Differences

Next shows two examples of certificate validation differences.

Example 1. Let a potential certificate chain be established. The certificate chain, as Figure 3 shows, contains a trust anchor, an intermediate certificate, and an end entity certificate. Let each certificate be in its validity period and have a correct signature.

This example triggers a validation difference during the CC phase. OpenSSL (ver. 1.1.1c) falsely accepts the end entity certificate, because name chaining can be performed and the certificate chain be successfully established and validated. GnuTLS, mbed TLS, NSS, and MatrixSSL reject the end entity certificate, because the end entity/intermediate certificate may not be parsed or the chain not be established, as (1) the subject name of a certificate, which "`MAY be carried in the subject field and/or the subjectAltName extension`" (Section 4.1.2.6, RFC 5280), should not be empty, and (2) the issuer field of a certificate "`MUST contain a non-empty X.500 distinguished name (DN)`" (Section 4.1.2.4, RFC 5280).

Example 2. Let a certificate with a CRL be validated and the established certificate chain contain a trust anchor and an end entity certificate, as Figure 4 shows. Let the certificate chain itself (without the CRL) be valid and accepted by all of the objective SSL/TLS implementations. The CRL

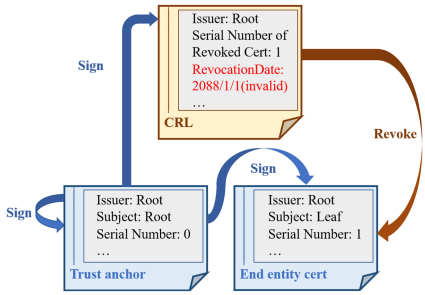


Fig. 4. A CRL and a certificate chain that trigger a validation difference. The root cause is that the CRL contains an invalid `RevocationDate`.

revokes the end entity certificate in this chain but contains an invalid date `RevocationDate` (e.g., a future date such as `2088/1/1`).

A CRL validation difference exists. Mbed TLS (ver. 2.16.2) and NSS (ver. 3.28.4) falsely accept the end entity certificate, because the CRL is invalid and should not revoke the end entity certificate. OpenSSL and GnuTLS reject the end entity certificate, because the date of revocation is not relevant to whether it has been revoked or not.

3 MUTATION AND DIFFERENTIAL TESTING

3.1 Certificate Mutation

We define 58 mutators (mutation operations) for supporting certificate mutation based on mucert's mutators in Reference [15]. transcert randomly picks a certificate (or a chain or a CRL) and mutates it, expecting the mutant to exploit some new validation policies in the validation code. As Table 1 shows, these mutators are classified into four categories.

(1) **Chain-specific mutator.** A chain-specific mutator is used to update a certificate chain, e.g., inserting one certificate into the chain or deleting one from the chain. A chain mutator is usually performed together with an updating of the issuers of the certificates on the chain such that each certificate is issued by the subsequent one.
(2) **Field-specific mutator.** A field-specific mutator is used to update a single certificate, e.g., rewriting the expiration date or modifying the signature algorithm. Note that when the subject of a certificate is updated, the issuer of the preceding certificate may be updated; when its issuer is updated, the subject of the subsequent certificate may be updated. We can also mutate a certificate by deleting one of its fields or adding one field.
(3) **Extension-specific mutator.** An extension-specific mutator is used to add, delete, or update an extension of a certificate, e.g., picking up a certificate's extension and changing its criticality.
(4) **CRL-specific mutator.** A CRL contains a list of serial numbers denoting a set of certificates to be revoked by this CRL. Correspondingly, we define (1) CRL-field-specific mutators that change the CRL fields and (2) revoked-cert-specific mutators that modify, add, or delete a revoked certificate in the list.

We adopt a grafting strategy when rewriting a certificate or its field. Given a certificate chain, we either (1) replace one certificate with an "invader" certificate that is randomly chosen from the certificate corpus, (2) insert the invader into the chain or use its field to update the corresponding

Table 1. Summary of the Certificate and CRL Mutators

Object	Category	Description
Certificate Mutators	Chain-specific	(1) Insert a certificate into a certificate chain; (2) Delete one or more certificates from a certificate chain; (3) Replace a certificate in a chain with another;
	Field-specific	(4) Modify a certificate's signature algorithm or field (e.g., subject, validity period, and so on); (5) Delete a certificate field; (6) Add a certificate field if it is `null`;
	Extension-specific	(7) Pick up a certificate and append a set of extensions to it; (8) Pick up a certificate and add a specific extension to it; (9) Pick up a certificate's extension and change its criticality; (10) Pick up a certificate's extension and change its value; (11) Pick up a certificate and delete one of its extensions; (12) Pick up a certificate and delete all of its extensions.
CRL Mutators	CRL-field-specific	(13) Modify a CRL's signature algorithm or field; (14) Pick up a CRL's extension and change its criticality or value; (15) Pick up a CRL and add one or more extensions to it; (16) Pick up a CRL and delete one or some of its extensions;
	Revoked-cert-specific	(17) Pick up a CRL and insert, replace or delete a revoked certificate in the list; (18) Pick up a CRL's revoked certificate and modify its field; (19) Pick up a CRL's revoked certificate and add one or more CRL-entry extensions; (20) Pick up a CRL's revoked certificate and delete one or some of its CRL-entry extensions; (21) Pick up a CRL-entry extension in a CRL's revoked certificate and change its value or criticality.

field of a certificate in the chain, or (3) choose one or more extensions of the invader certificate and add them into a certificate. This strategy helps produce syntactically correct mutants, otherwise they might have been rejected early during validation due to trivial parsing errors.

3.2 Differential Testing

Differential testing is a mature testing technology for large software systems [29, 40]: A test case is randomly generated, and output is compared for similar systems or systems having similar functionalities.

In our study, each test certificate, say, *cert*, is validated by the target SSL/TLS implementations and the validation results are then compared. As Table 2 shows, an SSL/TLS implementation may require two certificate files to be prepared when validating an end entity certificate: One contains a trusted CA certificate and another contains the end entity certificate; potential intermediate certificates are either in the trusted CA file, in the end-entity file, or in a certificate repository. A CRL file can also be included for checking whether the CRL validation runs normally.

A *validation difference* can then be revealed if a certificate *cert* is accepted by some implementations while rejected by the others. Furthermore, if a validation difference contains a combination of validation results (including the error messages) different from the others, then it is a *unique validation difference*.

4 COVERAGE DIRECTED CERTIFICATE GENERATION

4.1 Overview

We propose transcert, a coverage-directed approach to deeply testing X.509 certificate validation. Figure 5 shows the workflow of transcert.

Table 2. Prompts for Validating an X.509 Certificate *cert* against a Trusted CA Certificate *ca* and Some Validation Results

OpenSSL: openssl verify -verbose -CAfile $ca\ cert$
Some validation results:
(1) $cert$:ok.
(2) error 20 at 0 depth lookup: unable to get local issuer certificate. verification failed.
(3) error 18 at 0 depth lookup: self signed certificate. verification failed.
…
GnuTLS: certtool –verify –load-ca-certificate=ca < $cert$
Some validation results:
(1) Chain verification output: Verified. The certificate is trusted.
(2) Not verified. The certificate is NOT trusted. The certificate issuer is unknown.
(3) Not verified. The certificate is NOT trusted. The certificate issuer is not a CA.
…
mbed TLS: $cert\_app$ mode='file' filename=$cert\ ca\_file$=ca
Some validation results:
(1) ok.
(2) failed! The certificate is not correctly signed by the trusted CA.
(3) !mbedtls_x509_crt_parse failed to parse 1 certificates.
…
…

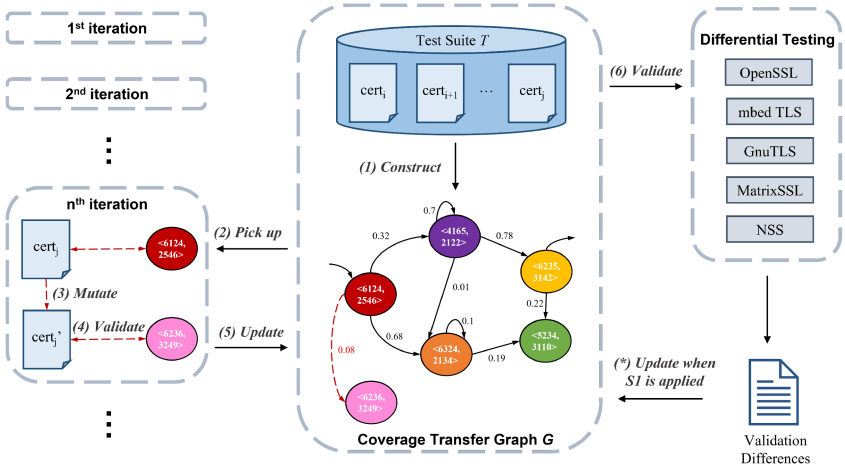


Fig. 5. Our transcert approach. Here (1) each vertex denotes an execution hit by a set of test certificates, (2) seeds can be selectively mutated for hitting new executions, and (3) once a new vertex (e.g., the pink vertex) is hit by some mutants, it is added to the coverage transfer graph.

Let T be a test suite containing a set of certificate seeds (and mutants). Let *validator* be a reference SSL/TLS implementation (e.g., OpenSSL). transcert directs the mutation process using coverage statistics: (1) pick up a certificate sample, say, *cert*, in T; (2) mutate *cert* to *cert′*; (3) validate *cert′* on *validator* and collect the coverage statistics; and (4) add *cert′* into T. The above steps are repeated, allowing T to be updated dynamically; a coverage transfer graph is constructed for directing the whole mutation process.

The certificate mutants are used to differentially test X.509 certificate validation. Any validation differences, when revealed, can be a hint for finding defect(s) in certificate validation in one or more SSL/TLS implementations.

Table 3. Lists of Functions and Their Descriptions

Function	Description
$f\_cov(cert_i)$	It computes the coverage value of $cert_i$ when it is validated on $validator$.
$f\_cert(u)$	It returns a set of certificates in T each of which, say, $cert$, has $u = f\_cov(cert)$.
$f\_trans(cert_i \rightarrow cert_j)$	It maps a mutation to a coverage transfer $u \rightarrow v$. Here $u = f\_cov(cert_i)$ and $v = f\_cov(cert_j)$. Besides, $f\_trans(u \rightarrow \bullet)$ and $f\_trans(\bullet \rightarrow v)$ are used to denote a set of coverage transfers from u and another set of transfers to v, respectively.
$f\_mutations(u \rightarrow v)$	It returns a set of mutations each of which, say, $cert_i \rightarrow cert_j$, has $u = f\_cov(cert_i)$, $v = f\_cov(cert_j)$. In particular, $f\_mutation(u \rightarrow \bullet)$ returns a set of mutations, each of which, say, $cert_i \rightarrow cert_j$, has $u = f\_cov(cert_i)$. Similarly, $f\_mutation(\bullet \rightarrow v)$ returns a set of mutations each of which, say, $cert_i \rightarrow cert_j$, has $v = f\_cov(cert_j)$.

4.2 Coverage-directed Certificate Generation

Despite strong efforts of randomly combining certificate fragments, the diversity of the generated certificate mutants needs to be measured and guaranteed. To address this problem, we introduce the notion of *coverage transfer graph*.

4.2.1 Coverage Transfer Graph. We use a *coverage transfer graph* to abstract certificate validations and mutation. A premise is that two X.509 certificates must achieve the same coverage information if they can traverse the same execution path. However, the two certificates are likely to traverse the same/similar execution paths if they have the same coverage information, letting some fine-grained coverage statistics (e.g., the statement and the branch coverage in this article) be collected.

As Figure 5 shows, a coverage transfer graph abstracts test certificates into a set of *coverage vertices* and certificate mutation into a set of *coverage transfers*. More formally,

Definition 4.1. A *coverage transfer graph* is a *weighted directed* graph $G = (V, E, W)$, where

- V is a set of n *coverage vertices*. Each vertex represents a unique coverage value and corresponds to a set of test cases hitting this value.
- E is a set of *coverage transfers* (i.e., edges) between the coverage vertices $E \subseteq \{u \rightarrow v | u, v \in V\}$. An edge $u \rightarrow v$ is established if some seed with a coverage value u is mutated to a mutant with v.
- W is a set of weights on the coverage transfers, each weight, say, $w = weight(u \rightarrow v)$, presenting a probability of transferring to v given u. Here the sum of all weights on the coverage transfers from u is 1, i.e., $\sum_{i=1}^{n} u \rightarrow V_i = 1$.

Let $cert_i$ be a certificate in the test suite T. Let $cert_i \rightarrow cert_j$ be a mutation, denoting that $cert_j$ is mutated from $cert_i$. We next define functions that map T to a coverage transfer graph G, and vice versa, as Table 3 shows.

Constructing an initial coverage transfer graph. In our study, each test certificate triggers an execution path when validated, which can be abstracted into a pair of statement and branch coverage values <$stmt$, $branch$>. Thus, given a set of certificate seeds, we validate them on $validator$ and collect their coverage statistics. The coverage transfer graph can be constructed by a set of vertices, each of which is associated with one or more certificates whose coverage values are the same as that represented in the vertex.

Updating a coverage transfer graph. Initially, G does not have any coverage transfers (i.e., edges). The coverage transfer graph can be explored and the coverage transfers be added during the mutation process.

Let a certificate $cert_i$ be chosen and mutated to $cert_j$. A vertex ($f\_cov(cert_j)$), if not exists, is added into G. A coverage transfer $f\_cov(cert_i) \rightarrow f\_cov(cert_j)$, if not exists, is established. The weights on the coverage transfers from $f\_cov(cert_i)$ also need to be updated. Let T_0 be a set of certificates holding the coverage value $f\_cov(cert_i)$. Let T_1 be a set of mutants mutated from certificates in T_0. Let $T_2 \subseteq T_1$ be a set of mutants holding the coverage value $f\_cov(cert_j)$. The weight on $f\_cov(cert_i) \rightarrow f\_cov(cert_j)$ is updated to $\frac{|T_2|}{|T_1|}$. The weights on the other coverage transfers are updated likewise.

Measuring the diversity. A coverage transfer graph can be further used to measure the diversity of a test suite and direct the mutation process. In our study, we compute the diversity of T using $|G.V|$, the number of vertices of G,

$$diversity(T) = \frac{|G.V|}{|T|} \times 100\%. \tag{1}$$

Obviously, the more vertices a coverage transfer graph has, the more execution paths T can traverse during testing; the diversity of T can decrease along with the increase of $|T|$. Thereafter, the goal of certificate generation is to design strategies to explore the coverage transfer graph, speeding up the growing of G along with the growing of the test suite, which will be explained in the following part.

4.2.2 Generating Certificates. The certificate seeds, and even the mutants, can be mutated. As Algorithm 1 shows, transcert iteratively mutates certificates. Each iteration first selects a vertex u from G and then picks up and mutates one certificate with the coverage u. The mutant is then validated and added into T; G is updated using the collected coverage statistics.

Sampling on the coverage transfer graph. Certificates need to be selectively mutated on the basis of their coverage statistics. In particular, some vertices need to be more frequently chosen than the others, since the corresponding execution paths are less traversed and/or more easily transferred to new execution paths.

We select vertices (with their certificates) by a potential function:

$$potential(u) = \frac{|f\_trans(u \rightarrow \bullet)|}{|f\_mutation(u \rightarrow \bullet)|}. \tag{2}$$

Here the potential of a vertex relies on how many of its certificates have been mutated and how many coverage transfers (i.e., edges) the vertex has. The less a vertex has been chosen for mutation and the more coverage transfers, the higher the potential.

Intuitively, the higher a vertex's potential, the more likely its certificates need to be mutated. A vertex's potential decreases after its certificates are mutated. Thus, we follow the approach in Reference [14] to sample vertices by letting probabilities meeting a *geometric distribution*. Let $Array$ be an array where all vertices in G be stored in descending order of their potential values. Let the kth element in $Array$ be chosen with probability $(1-p)^k p$, where p is the success probability. Let $(1-p)^{Array.size} = \varepsilon$, where ε is a very small value.

A vertex is then chosen from $Array$ using $rand$, a random real value between 0.0 and 1.0. Given a random real value, the kth element in $Array$ is chosen

$$k = \lfloor \log_{\varepsilon} (1 - rand)^{Array.size} \rfloor. \tag{3}$$

A vertex's certificates can be further chosen by random for the next mutation.

ALGORITHM 1: transcert algorithm to generate certificates. This algorithm employs a coverage graph G to direct certificate mutation, and updates G using the coverage statistics collected.

Input: *iter*: the number of iterations; $SEED$: a set of certificate seeds
Output: $MUTANT$: a set of certificate mutants

```
Function Main():
    T ← SEED;
    Create a coverage transfer graph G for T;
    for i : 1 to iter
        /* (1) Next picks up a certificates and mutates it                    */
        u ← Pickup − A − Vertex − Sample(G);
        for each certificate cert in f_cert(u)
            if (u == f_cov(cert))&&(cert.mutable == true) :
                CERT ← CERT ∪ {cert};
        cert_i ← Random.Choice(CERT);
        cert_j ← Mutate(cert_i);
        /* (2) Next updates T                                                 */
        cert_j.mutable ← true;
        if the strategy S1 is used :
            On − the − Fly − Testing(cert_j);
        T ← T ∪ {cert_j};
        /* (3) Next updates G                                                 */
        Validate cert_j on validator and collect coverage statistics;
        if f_cov(cert_j) is not in G :
            G.Add(f_cov(cert_j))
        if f_cov(cert_i) → f_cov(cert_j) is not in G :
            G.Add(f_cov(cert_i) → f_cov(cert_j));
        Update weights on the coverage transfers f_trans(u → •);
    MUTANT ← T − SEED;
    Return MUTANT;
Function Pickup-A-Vertex-Sample(G):
    /* Sort G's vertices using their potentials                               */
    M ← Sort(G.V);
    rand ← new Rand();
    k ← ⌊log_ε(1 − rand)^size⌋;
    Return M[k];
Function On-the-Fly-Testing(cert):
    Validate cert on different SSL/TLS implementations;
    if validation difference exists :
        cert.mutable = false;
```

4.3 Optimization

We design three strategies for optimizing the iterative process of transcert: S1 is taken to reduce the redundancy of validation differences, S2 is to deeply test certificate validation, and S3 is to simplify the mutation process (including sampling and accepting/rejecting the mutants).

S1. Cutting propagation. Differences can be propagated, making a number of redundant differences exist: If a certificate can expose a validation difference, then its mutant(s) are more likely to expose the same validation difference.

To cut the propagation of validation differences, we perform the on-the-fly testing of certificates on the target SSL/TLS implementations. A certificate is *unmutable* if it can trigger any validation

difference. As Algorithm 1 shows, during the mutation process, an unmutable certificate will not be chosen for further mutation.

Note that although S1 helps refrain differences from being propagated, the on-the-fly differential testing of certificate validation can definitely increase the cost of test generation. In addition, it requires all of the target certificate validation implementations to be ready during certificate generation, which may be impractical when an implementation is under development.

S2. Performing extension-specific mutation. The mutators mentioned above are not effective in deeply test certificate validation. Many of differences are raised, because implementations are differently fault-tolerant in parsing illegal certificates. Chain validation, which is the most security-critical component in certificate validation, is still not thoroughly tested, because a chain may not even be constructed between the certificate mutant and the CA.

S2 performs extension-specific mutation on an established chain. Considering that name matching plays the key role in chain construction, we keep the names of all certificates on a chain (including the subjects and the issuers) unchanged but apply the extension-specific mutators to mutate these certificates. RFC 5280 defines 15 standard extensions and 2 private Internet ones [18]. The extension-specific mutators, which are shown in Table 1, change certificate extensions. Thus certificate mutants can carry various combinations of extra information and constraints (e.g., maximal chain length and name constraints) that may be differently processed by the CC and CV components of certificate validation.

S3. Using accumulative code coverage (ACC) to direct certificate mutation. Inspired by mucert [15], we supplement transcert with a strategy that employs ACC to direct certificate mutation. This strategy allows us to cast the certificate mutation problem into an optimization problem. Let $Cov(Cert)$ be defined for computing the accumulative code coverage of a test suite $Cert$ w.r.t. a reference SSL/TLS implementation:

$$Cov(Cert) = Cov(cert_0) \oplus \cdots \oplus Cov(cert_n), \tag{4}$$

where $Cov(cert)$ denotes the coverage achieved by an individual certificate cert, and $\oplus$ allows the coverage to be computed cumulatively.

Our setting helps the test suite accept test certificates distinct from existing ones. Let $cert$ in $Cert$ be mutated to $cert'$ in $Cert'$. Let $Cov(Cert) < Cov(Cert')$. The certificate $cert'$ is obviously distinct from $cert$ or any other certificates in $Cert'$, as it exploits new validation code (or branches). Therefore, the S3 strategy casts the problem of certificate mutation to an optimization problem: Given the test suite $Cert$, how can its certificates be mutated continuously such that

$$Cov(Cert) \ll Cov(OptimizedCert) \leq \top, \tag{5}$$

where $Cov(Cert) \ll Cov(OptimizedCert)$ denotes that the coverage should be increased as much as possible and $\top$ is an upper bound that may be achieved?

The transcert algorithm is then redesigned as follows. It first selects a test suite of n certificates (certificate chains more rigorously) and then performs a number of iterations. Each iteration chooses exactly one certificate and mutates it. The certificate mutant is accepted if it leads to an increase of the accumulative code coverage w.r.t. the reference SSL/TLS implementation; otherwise, it can be accepted with a very low probability.

5 EVALUATION

We implement transcert and evaluate it against frankencert [10], NEZHA [43], and RFCcert [50]. The evaluation is designed to answer the following research questions:

Table 4. Test Suites Generated

	Test Suite	Technique and Strategies
Group 1	$TCERT_0$	transcert
	$TCERT_1$	transcert (S1)
	$TCERT_2$	transcert (S1, S2)
	$TCERT_3$	transcert (S3)
	$FCERT$	frankencert
	$NEZHA$	transcert with NEZHA's δ-diversity
Group 2	$TCERT_0(30K)$	transcert (with 30K iterations)
	$FCERT(30K)$	frankencert (with 30K iterations)
	$NEZHA(30K)$	transcert with NEZHA's δ-diversity (with 30K iterations)
Group 3	$RCERT$	RFCcert (with non-randomized certificates)
Group 4	$TCERT_4$	transcert (with CRL mutators)

$TCERT_0$, $TCERT_1$, $TCERT_2$, $TCERT_3$, $NEZHA$, $FCERT$, and $RCERT$ are test suites generated for testing path validation, and $TCERT_4$ are the one created by CRL mutators for testing CRL validation.

RQ1. (Sufficiency) How sufficient are the transcert certificates in testing X.509 certificate validation?

RQ2. (Diversity) How diverse are the transcert certificates?

RQ3. (Effectiveness&Efficiency) How effective and efficient are the transcert certificates in revealing X.509 certificate validation differences among SSL/TLS implementations?

RQ4. (Conformance to Validation Rules) How effective is transcert in conformance testing of certificate validation implementations to RFC 5280 compared with the RFC-directed technique RFCcert?

5.1 Experimental Setup

Seeds. We evaluate transcert using seeds from two sources. (1) We adopt frankencert's corpus, which contains 1,005 certificates. These certificates are collected by employing Zmap to scan the Internet on Port 443 and establishing SSL connections [10]. (2) We additionally collect 5,180 CRL files, of which 500 are used as seed CRLs and the others as a corpus for mutation.

We use OpenSSL to generate a self-signed certificate as the root CA certificate. We re-sign the certificate (or CRL) seeds using the root CA certificate, ensuring that these seeds did not trigger any validation difference among the target validation implementations.

Test suites. We run the tests on OpenSSL and use gcov, the GNU coverage test tool,[7] to collect the coverage statistics. During each iteration, gcov creates a logfile indicating how many times each line/branch of the source code of OpenSSL has executed.

We produce 11 test suites, as Table 4 shows, and make comparisons. The iteration number of each approach except *RCERT* is set as 10K or 30K. We repeat each approach (with its optimization strategies) 10 times and calculate the averages and standard deviations of every statistics for comparison. Here $TCERT_4$ is specifically generated for testing CRL validation; the other test suites are comparable, since they are generated only for testing path validation. We re-implement NEZHA, which integrates transcert's mutators and NEZHA's δ-diversity metric for making comparisons, because the source code of the NEZHA tool relies on an outdated LibFuzzer library and is no longer supported on current platform. In addition, *RCERT* contains 89 certificates created by the developers of RFCcert.

[7] https://gcc.gnu.org/onlinedocs/gcc/Gcov-tool.html.

Metrics. Let T be a test suite. We evaluate the approaches using the following metrics:

- *ACC*. We compute the accumulative statement/branch coverage achieved by T. The higher the *ACC* value, the more effective the test suite is expected to be.
- ***Diversity (DIV)***. We count the numbers of coverage vertices and coverage transfers and then compute the diversity of T.
- *Precision*. Let $VDIF$ be a set of validation differences revealed by T and $UDIF$ a set of unique ones (see Section 3.2). We count how many validation differences and how many unique ones can be revealed by T, respectively. *Precision*, which indicates how effective is T in exposing validation differences (and potential defects), can be calculated by

$$Precision = \frac{|VDIF|}{|T|} \times 100\%. \tag{6}$$

Target SSL/TLS implementations and platform. We validate the certificate mutants on five SSL/TLS implementations: OpenSSL (ver. 1.1.1c), mbed TLS (ver. 2.16.2), GnuTLS (ver. 3.6.8), NSS (ver. 3.28.4), and MatrixSSL (ver. 4.2.1). Certificate validation in other SSL/TLS implementations can also be tested using the test suites generated.

5.2 RQ1: Sufficiency

The six certificate test suites with 10K iterations are of different sizes. $TCERT_0$, $TCERT_1$, $TCERT_2$, and *FCERT* contain 10,000 mutants, since each iteration adds one mutant into a test suite. On the contrary, $TCERT_3$ contains only 1,005 certificates, as each iteration replaces at most one certificate in the test suite with a mutant. *NEZHA* generates 98.3 certificates on average, far less than other test suites, since it only keeps certificates triggering new discrepancies. It leads to an illusion that $TCERT_3$ and *NEZHA* might be more efficient than the other test suites in employing a small test suite to test X.509 certificate validation.

The evaluation results show that the six test suites in Group 1 are not of significant differences in cumulatively covering the source code of OpenSSL. As Table 5 shows, $TCERT_0$ and $TCERT_1$ achieve a little higher code coverage (54.0~301.4+ statements and 49.9~172.7+ branches on average) than $TCERT_3$, *NEZHA*, and *FCERT*, even if $TCERT_3$ takes the ACC value as the guidance. Since $TCERT_3$ has only 1,005 certificates in our study, it is not sufficient in testing certificate validation. $TCERT_2$ achieves the lowest statement/branch coverage values among $TCERT_{1\sim3}$. The reason is that the strategy S2 only updates the extensions of certificates, concentrating only on the scenarios when certificate chains has been constructed; certificate parsing/chain construction algorithms are insufficiently tested by $TCERT_2$.

Figure 6(a) shows an increase of the ACC value of $TCERT_0$ during the mutation process. Indeed, $TCERT_0$ has already achieved higher statement coverage than *FCERT* and $TCERT_3$ after 1,000 iterations. It clearly indicates that transcert is competent enough in covering the source code of certificate validation.

Answer to RQ1: The six test suites in Group 1 are not of significant differences in covering certificate validation. Indeed, $TCERT_0$ and $TCERT_1$ can achieve higher code coverage.

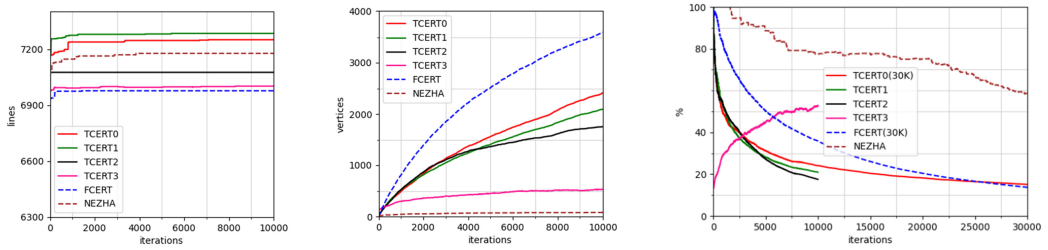
5.3 RQ2: Diversity

As Table 5 shows, the diversity of $TCERT_3$ is 54.17%, much higher than the three transcert test suites and *FCERT*. Meanwhile, as what we explained in Section 5.2, it can be an illusion: As $TCERT_3$ does not increase its size during iterations, its diversity can be higher and higher, while it can cover at

Table 5. Comparison of transcert against frankencert, NEZHA, and RFCcert

# Averages *(#) Standard deviations*	$\|T\|$	*ACC* (stat.)	*ACC* (branch)	$\|G.V\|$	$\|G.E\|$	*DIV*	$\|VDIF\|$	$\|UDIF\|$	*Precision*
FCERT	10,000	6,961.2	3,467.0	3,613.0	/	36.13%	434.1	6.0	4.34%
	/	*(20.6)*	*(17.1)*	*(17.1)*	/	*(0.17%)*	*(36.3)*	*(0.0)*	*(0.36%)*
NEZHA	98.3	7,189.3	3,549.0	80.0	/	81.74%	71.7	49.7	72.99%
	(7.9)	*(5.8)*	*(5.4)*	*(1.6)*	/	*(4.80%)*	*(4.5)*	*(1.7)*	*(1.79%)*
$TCERT_0$	10,000	7,243.3	3,598.9	2,428.0	6,280.8	24.28%	4,432.9	70.9	44.33%
	/	*(19.7)*	*(16.4)*	*(56.5)*	*(47.3)*	*(0.56%)*	*(177.3)*	*(9.3)*	*(1.77%)*
$TCERT_1$	10,000	7,262.6	3,615.2	2,074.0	5,590.0	20.74%	4,196.5	74.1	41.97%
	/	*(23.8)*	*(19.7)*	*(17.0)*	*(52.0)*	*(0.17%)*	*(97.5)*	*(7.7)*	*(0.98%)*
$TCERT_2$	10,000	7,077.5	3,442.5	1,649.3	6,290.0	16.49%	2,739.8	25.4	27.40%
	/	*(4.2)*	*(5.8)*	*(53.1)*	*(59.7)*	*(0.52%)*	*(89.3)*	*(3.0)*	*(0.89%)*
$TCERT_3$	1,005	7,031.8	3,486.8	544.4	/	54.17%	581.0	30.8	57.81%
	/	*(36.9)*	*(22.3)*	*(15.2)*	/	*(1.51%)*	*(50.0)*	*(1.2)*	*(4.98%)*
FCERT	30,000	6,987.6	3,474.0	4,137.8	/	13.79%	1,406.1	6.0	4.69%
(30*K*)	/	*(7.8)*	*(17.1)*	*(66.7)*	/	*(0.22%)*	*(40.7)*	*(0.0)*	*(0.14%)*
NEZHA	172.5	7,198.6	3,565.1	125.4	/	73.05%	124.3	75.5	72.21%
(30*K*)	*(11.5)*	*(18.0)*	*(17.4)*	*(2.5)*	/	*(5.45%)*	*(2.9)*	*(2.2)*	*(2.71%)*
$TCERT_0$	30,000	7,246.7	3,603.7	4,513.3	17,245.3	15.04%	7,858.5	108.0	26.20%
(30*K*)	/	*(5.4)*	*(5.4)*	*(87.1)*	*(29.9)*	*(0.29%)*	*(33.5)*	*(4.9)*	*(0.11%)*
RCERT	89	6653	3184	40	/	44.94%	29	10	32.58%
	/	/	/	/	/	/	/	/	/

Here T is the test suite and G is the coverage transfer graph of T. Here the coverage transfers w.r.t. $TCERT_3$, *NEZHA*, *FCERT*, and *RCERT* are not collected, as the mutation activities in the test suites are not kept in the test suites; $TCERT_4$ is omitted, since it is specially designed for testing CRL validation.



(a) ACC (b) G.V (c) DIV

Fig. 6. Changes of ACC values, $|G.V|$ values, and diversities of test suites during the iterations.

most 1,005 test scenarios (in the study 559). Similarly, *NEZHA* only generates an average of 98.3 certificates during 10K iterations, achieving a high diversity (81.74%).

FCERT has higher diversity than $TCERT_0$, $TCERT_1$, and $TCERT_2$, indicating that the random crossovering strategy is very effective in diversifying a test suite. However, as Figure 6(b) and (c) show, although the diversity of $TCERT_0$ is lower than *FCERT* and their diversities decrease during certificate generation, that of $TCERT_0$ decreases more slowly along with the increase of mutants after 8,000 iterations.

The above findings strongly indicate that ACC is not a good metric for improving the diversity of a test suite. A higher ACC value does not definitely indicate a higher diversity; the ACC value usually increases along with the growing of a test suite, while the diversity decreases.

We run transcert and frankencert for another 20,000 iterations, creating another two test suites, $TCERT_0(30K)$ and $FCERT(30K)$, for make further comparisons. Indeed, the diversities of the two

test suites meet at around 26,000 iterations. After that, the diversity of $TCERT_0$ becomes a little higher than *FCERT*.

The optimization strategies, S1 and S2, reduce, rather than raise, the diversities of the test suites. $TCERT_0$, $TCERT_1$, and $TCERT_2$ hit 2,428.0, 2,074.0, and 1649.3 coverage vertices on average, respectively. S1 reduces the diversity, because it cuts propagation of validation differences—the mutable certificates in $TCERT_1$ are far less than those in $TCERT_0$, reducing the potential coverage transfers and new coverage vertices during iterations. $TCERT_2$ hits the fewest coverage vertices, because the source code w.r.t. the CP/CC components is insufficiently covered, and thus the various combinations of the policies for parsing and processing certificates are less likely covered during testing.

Answer to RQ2: The diversities of $TCERT_3$ and *NEZHA* are higher than those of $TCERT_0$ and $TCERT_1$ because of their small size; *FCERT*'s diversity can decrease, and be lower than that of $TCERT_0$ after 26,000 iterations. In addition, ACC is not appropriate for improving the diversity of a test suite.

5.4 RQ3: Effectiveness and Efficiency

Effectiveness. All of the test suites reveal validation differences and unique ones. As Table 5 shows, *FCERT*, *NEZHA*, $TCERT_3$, and *RCERT* can reveal 434.1, 71.7, 581.0, and 29 validation differences on average, respectively. Comparatively, the other three transcert test suites are more effective in revealing validation differences—$TCERT_0$, $TCERT_1$, and $TCERT_2$ can reveal 4,432.9, 4196.5, and 2739.8 validation differences, respectively. Note that a high number of validation differences does not mean a high number of unique validation differences, since validation differences can be propagated. For instance, $TCERT_0$ can find more validation differences than $TCERT_1$ but contain less unique ones; $FCERT(30K)$ reveals one unique difference per 234 validation differences on average, while $NEZHA(30K)$ reveals a unique one every 2. On the contrary, $TCERT_1$ can find more unique ones from fewer validation differences by cutting the propagation of validation differences using the on-the-fly testing strategy.

transcert takes a stronger strategy in revealing unique validation differences than the other techniques. First, although *FCERT* can hit many more coverage vertices than $TCERT_0$ and $TCERT_1$, it is not very effective in uncovering unique validation differences. Indeed, many coverage vertices correspond to the scenarios that the certificates can consistently pass or fail in certificate parsing. Second, transcert(S3), using ACC to direct certificate mutation, achieves the higher precision than transcert: It hits 581.0 validation differences and 30.8 unique ones on average using 1,005 test certificates. However, each of the approaches mutates the certificates for 10,000 iterations; the more numbers of validation differences and unique ones revealed by the transcert test suites can definitely help reveal the more defects in SSL/TLS implementations. In fact, transcert ($TCERT_0$) outperforms frankencert and transcert(S3) by 12× and 2.3×, respectively. Third, coverage transfer graph outperforms δ-diversity in exploring unique test certificates. Although *NEZHA* achieves the highest precision among the techniques due to the fact that it retains a small number of certificates, yet, during 10K or 30K iterations, $TCERT_0$ can reveal 1.4× and 7× unique differences than *NEZHA* and *RCERT*, respectively, which clearly shows the effectiveness and efficiency of using coverage transfer graph in exploring the input space, respectively.

In conclusion, $TCERT_0$ and $TCERT_1$ reveal the more numbers of unique validation differences than *FCERT*, *NEZHA*, $TCERT_3$, and *RCERT*: $TCERT_0$ (or $TCERT_1$) catches 15 (or 18) validation differences that *FCERT*, *NEZHA*, $TCERT_3$, and *RCERT* cannot catch, while only misses three validation difference that they can catch. $TCERT_2$ reveals seven unique validation differences that are

Table 6. Efficiency of Each Approach

(a) Time cost. Each iteration of frankencert and NEZHA takes much shorter time than transcert and transcert(S3), as they do not collect coverage statistics during certificate generation.

	frankencert	NEZHA	transcert	transcert(S3)
Cost per certificate (sec.)	0.004	41.38	7.98	11.93
Cost per testing (sec.)	0.03	0.03	0.03	0.03
Cost per iteration	0.034	0.52	8.12	11.96

(b) Effectiveness within one hour. Here: (1) frankencert generates the most certificates; (2) frankencert triggers the most validation differences, while transcert and NEZHA much more unique ones; and (3) $|G.V|$ w.r.t. frankencert is not calculated due to the high cost in collecting the mutants' coverage statistics.

	frankencert	NEZHA	transcert	transcert(S3)
iteration	105,882	7,288	443	301
$\|G.V\|$	/	74	325	303
$\|VDIF\|$	4,609	58	176	81
$\|UDIF\|$	6	40	31	11

specific to the CV components of certificate validation. The other test suites, including $TCERT_0$ and $TCERT_1$, are less effective in revealing these validation differences.

Efficiency. As Table 6(a) shows, frankencert is much faster in generating certificates. It only randomly combines certificate seeds, generating 15K bogus certificates within 60 seconds. On the contrary, transcert need to collect coverage statistics during each iteration, which slows down the certificate generation process significantly: transcert generates 7.58 certificates within 60 seconds, and transcert(S3) 5.02 certificates. transcert outperforms transcert(S3) in that it omits merging the coverage statistics of the whole test suite. NEZHA does not collect coverage statistics, and thus the time cost per iteration is also less than that of transcert. RFCcert needs time cost for collecting and formalizing RFC rules. Meanwhile, the time cost per certificate is still higher, since NEZHA does not generate a certificate during each iteration. However, all three approaches are fast in validating each mutant on the five SSL/TLS implementations—on average 0.03 seconds is spent on using each mutant to differently test the five implementations.

Let frankencert, NEZHA, transcert(S1), and transcert(S3) spend the same time (1 hour) on certificate generation and differential testing. As Table 6(b) shows, frankencert, NEZHA, transcert(S1), and transcert(S3) complete 105,882, 7,288, 301, and 443 iterations, hitting 4,609, 58, 176, and 81 validation differences, respectively. transcert(S1) outperforms frankencert and transcert(S3) by 5× and 3×, respectively, in revealing unique validation differences.

Answer to RQ3: $TCERT_0$ reveals 71 unique validation differences on average, outperforming *FCERT*, $TCERT_3$, *NEZHA* and *RCERT* by 12×, 2.3×, 1.4× and 7×, respectively. transcert outperforms frankencert by 5× in revealing unique validation differences within one hour.

5.5 RQ4: Conforming to Validation Rules

We compare transcert against RFCcert, investigating the conformance of the generated certificates to validation rules. RFCcert is a state-of-the-art approach, advocating an idea of using RFC rules to direct certificate generation followed by differentially testing SSl/TLS implementations. Its

workflow contains three steps: (1) automatically extract validation rules from RFCs; (2) construct test certificates each of which deliberately violates or meets one specific RFC rule; and (3) use test certificates to differentially test SSL/TLS implementations and check whether RFC rules are supported by these implementations gracefully.

Validation rules. We extract 459 validation rules from RFC 5280. These rules, as Table 7 shows, specify the syntactical/semantic constraints on a legal certificate (and its fields), a CRL file, a certificate chain, or the whole validation process. Table 7 classifies the validation rules from four respects: (1) PRODUCER, CONSUMER, and SHARED rules. Rules are classified according to the compliance objects; (2) BREAKABLE and UNBREAKABLE rules. Rules are classified, indicating whether they can be violated; (3) MUST, SHOULD, MAY, UNDEFINED, and ASN.1 rules. Rules are presented in natural language or the ASN.1 grammar; they thus can be classified by their representations and the modal verbs the rules use. Note that UNDEFINED rules are recognized in our study, since they can still be used in certificate validation, while they are not covered by RFCcert; (4) CERTIFICATE VALIDATION and CRL VALIDATION rules.

RFCcert extracts a set of well-defined RFC rules for directing certificate generation. Meanwhile, there exist threats to rule extraction, making RFCcert fail in hitting some validation rules during certificate generation:

(1) *RFCcert may miss catching some validation rules* if they do not contain modal verbs or are defined in other RFCs (some are even not defined in any RFCs). For instance, *Example 2* in Section 2.3, which shows a certificate with a future `RevocationDate` field, is not related with any rule explicitly specified in RFCs.
(2) *A rule can be described in different parts in RFC 5280.* For example, in the rule "`Certificate users SHOULD be prepared to gracefully handle such certificates,`" "`such certificates`" are declared in another statement: "`non-conforming CAs may issue certificates with serial numbers that are negative or zero.`"
(3) *Rules may be misclassified.* For instance, the rules about the extensions' criticality can be misclassified as PRODUCER rules and omitted by RFCcert.

The rules used in our study are extracted on the basis of the rules provided by RFCcert. Meanwhile, we add more rules into the set if our test certificates reveal that they are useful in certificate validation.

Rule Conformance. Test certificates are designed for testing the conformance of certificate validation implementation(s) to specific validation rules. Let C be a test suite. We use $Rule(C)$ to denote a set of objective validation rules to be tested by the certificates in C.

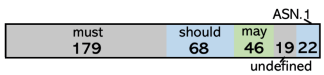
An investigation of the rules covered by $RCERT$, $TCERT_0$, and $TCERT_4$ reveals that

- *RFCcerts and transcerts are designed for different objective rules.* Table 8 shows that transcerts correspond to more validation rules than RFCcerts (120 vs. 86).[8] Figure 7 shows the differences between transcerts and RFCcerts in covering path validation rules. In particular, transcerts cover 85 rules uncovered by RFCcerts, because they have different objective rules. *First*, RFCcert mainly tests SHARED and CONSUMER rules, while transcert also tests PRODUCER rules. *Second*, $Rule(RCERT)$ contains mainly BREAKABLE rules, while $Rule(TCERT_0)$ also contains many UNBREAKABLE rules. *Third*, in addition to the MUST, MAY, and SHOULD rules, $Rule(TCERT_0)$ contains many UNDEFINED and ASN.1 rules.

  Section 5 of RFC 5280 is related to CRL rules. These rules define multiple constraints on CRL Fields, CRL Extensions, and CRL Entry Extensions. $Rule(RCERT)$ and $Rule(TCERT_{0\sim3})$

[8] In Reference [50], the 89 RFCcerts are designed for testing 69 validation rules in RFC 5280.

Table 7. Validation Rules in RFC 5280

	Proportion	Example Rule
Certificate validation (334 rules extracted from Sections 4, 6~8, RFC 5280)		**PRODUCER**: *Conforming CAs* MUST NOT use serialNumber values longer than 20 octets. (Section 4.1.2.2) **CONSUMER**: *Implementations* SHOULD be prepared to accept any version certificate. (Section 4.1.2.1) **SHARED**: If only basic fields are present, the version MAY be 2 or 3. (Section 4.1.2.1)
		**BREAKABLE**: GeneralizedTime values MUST NOT include fractional seconds. (Section 4.1.2.5.2) **UNBREAKABLE**: A non-critical extension MUST be processed if it is recognized. (Section 4.2)
		**MUST**: Certificate users *MUST* be able to handle serialNumber values up to 20 octets. (Section 4.1.2.2) **UNDEFINED**: If (pathLenConstraint) present, asserts the keyCertSign bit. (Section 4.2.1.3) **ASN.1**: Certificate ::= SEQUENCE { tbsCertificate TBSCertificate, signatureAlgorithm AlgorithmIdentifier, signatureValue BIT STRING } (Section 4.1)
CRL validation (125 rules extracted from Sections 5~8, RFC 5280)		**PRODUCER**: *Conforming CRL issuers* MUST include the nextUpdate field in all CRLs. (5.1.2.5) **CONSUMER**: This specification RECOMMENDS that *implementations* recognize this extension. (Certificate Issuer). (Section 5.3.3) **SHARED**: (Authority Information Access) MUST be marked as non-critical. (Section 5.2.7)
		**BREAKABLE**: Conforming CRL issuers MUST include this extension (Authority Key Identifier) in all CRLs issued.(Section 5.2.1) **UNBREAKABLE**: This specification RECOMMENDS that implementations recognize this extension. (Certificate Issuer). (Section 5.2.3)
		**SHOULD**: This specification *RECOMMENDS* that implementations recognize this extension. (Certificate Issuer). (Section 5.3.3) **MAY**: The CRL issuer *MAY* also generate delta CRLs. (Section 5) **ASN.1**: CRLNumber ::= INTEGER (0..MAX) (Section 5.2.3)

We have slightly more validation rules than [50], as we also include some UNDEFINED and implicit rules.

Table 8. Comparison of transcert and RFCcert in Covering Validation Rules

	Certificate Validation Rules Covered	CRL Validation Rules Covered
$RCERT$	86	/
$TCERT_0$	89	/
$TCERT_4$	/	31

do not contain CRL rules. Comparatively, $Rule(TCERT_4)$, contains 24 BREAKABLE and 7 UNBREAKABLE CRL rules, most of which are MUST rules, except four SHOULD rules and two UNDEFINED rules.

- *A rule can be complicated, containing multiple rule fragments.* A rule may not be sufficiently tested by one certificate meeting the rule and another violating it. Figure 8 shows a rule with the condition "$CA = True \ \& \ KeyCertSign \in KeyUsage$"; four or more test certificates need to be designed for testing this condition.
- *A test scenario may involve multiple rules, while RFCcert constructs each certificate for meeting/violating a single rule.* transcert randomly mutates seeding certificates, undeliberately

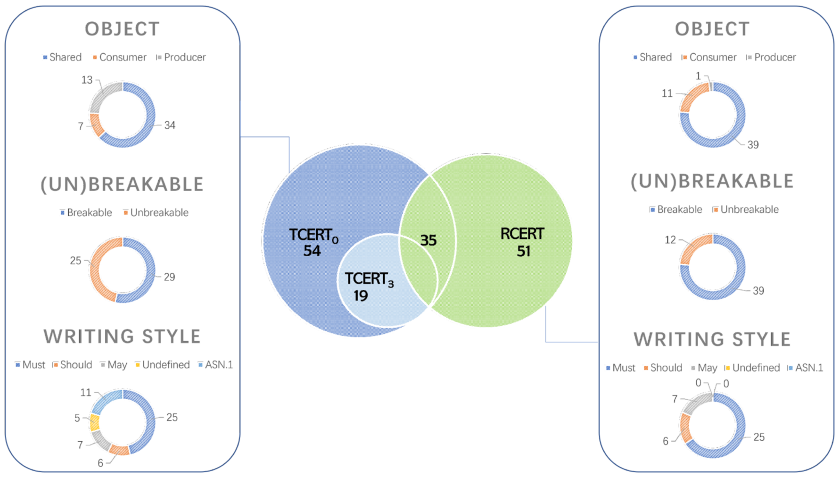


Fig. 7. Analysis of the certificate validation rules covered by *TCERT*, *RCERT*, or both. The pie charts show the proportion of each type of rules uniquely covered by $TCERT_0$ or *RCERT*.

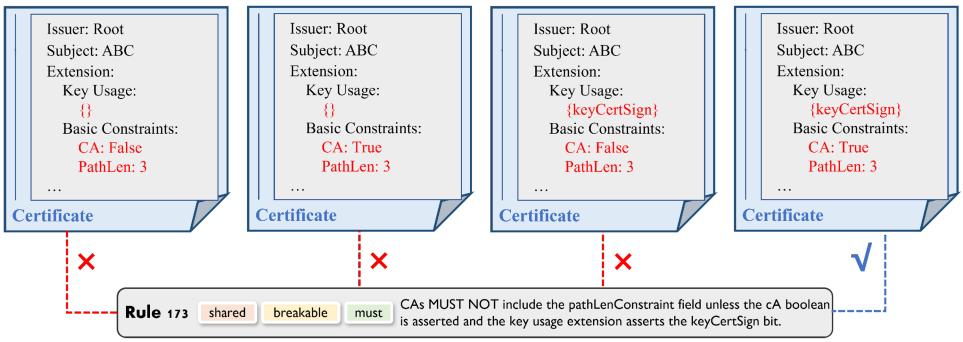


Fig. 8. Four test certificates designed for testing a rule of multiple rule fragments. Here ✓ and × indicates the certificate meets and violates the rule, respectively.

covering validation rules and their combinations. Figure 9 shows that a transcert-generated certificate chain (i.e., *Example 1* in Section 2.3) violates three rules and meet one rule. With a sufficient number of mutations, transcert can generate certificates covering different rules and their combinations. Comparatively, RFCcert fails in combination testing of validation rules.

- *Others.* RFCcerts and transcerts are different in many other respects. For example, RFCcerts are created only by meeting explicit rules, while transcerts may meet/violate implicit rules. The length of all certificate chains constructed by RFCcert is 2, while transcert can have a chain of 100 certificates.

Only 120/459 rules are covered by transcert in our study. A detailed analysis of the 339 rules missed by transcert reveals the following:

- One hundred sixty-five rules are missed because of the absence of specific fields and/or field values in seed certificates. For instance, the seeds of transcert do not contain any certificate with `NameConstraints` extension, and thus all rules related to this extension are missed by transcert.

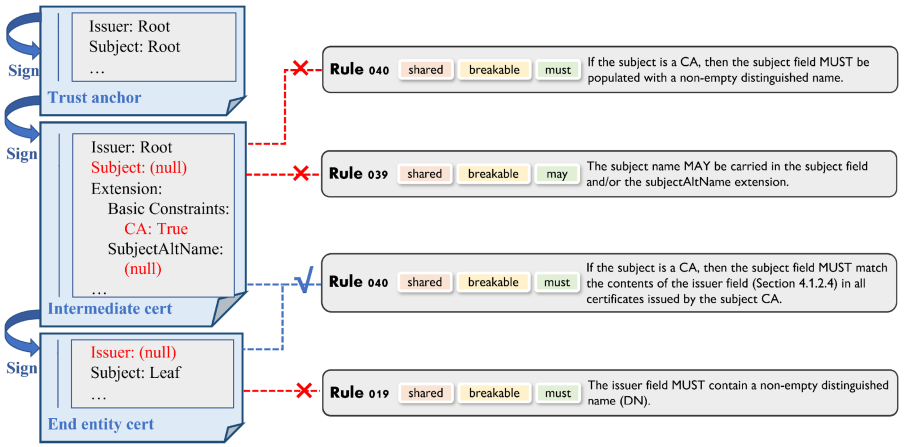


Fig. 9. A certificate chain that meets some validation rules and violates some others. Note that ✓ means the conformance of a certificate to a rule and × a rule violation.

- Fifty-four rules are not covered, because transcert built atop PyOpenSSL does not support for some field modifications. For example, testing of the rule "`this field (signatureAlgorithm) MUST contain the same algorithm identifier as the signature field in the sequence tbsCertificate`" needs a certificate with two different algorithm identifier fields; PyOpenSSL can only feed identical identifiers into a certificate.
- Ninety-one rules are not tested by the file-mode validation of SSL/TLS implementations. For example, the rule "`The scope of a delta CRL MUST be the same as the base CRL that it references`" is missed, because the file-mode cannot simultaneously validate two CRLs (the delta and the base). Besides, the rules related to Client-Sever mode validation are also not covered in our study.
- Twenty-nine rules are missed for some other reasons: (1) the expression is vague, e.g., "`A conforming implementation MAY also support validation with respect to some point in the past`"; (2) some rules define implementation details and can only be validated by checking the code of SSL/TLS implementations, e.g., "`Conforming implementations MAY impose requirements on application specific extension`"; and (3) a few rules are not relevant to the validation process, e.g., "`CA issuer names SHOULD be formed in a way that reduces the likelihood of name collisions.`"

Answer to RQ4: transcert supports testing 120 objective validation rules in RFC 5280 [18], 85 of which are not covered by RFCcerts; transcerts can undeliberately cover many test scenarios missed by RFCcerts. However, 339 of 459 rules are missed by transcerts; these rules can be covered if transcert takes certificates with rich fields/extensions as seeds, employs more powerful SSL/TLS libraries to implement mutators, or let users be involved in certificate generation.

5.6 Threats to Validity

Next discusses the internal and external validity of our findings together with possible threats to them.

Table 9. The p-values of the Two-Tailed Mann–Whitney U-tests and the Standardized Effect Size $\hat{A}_{12}$ for the Results of *ACC* (statement), *ACC* (branch), and $|UDIF|$ shown in Table 5

Pairwise Comparison	*ACC* **(statement)**	*ACC* **(branch)**	$\lvert UDIF \rvert$
$TCERT_0$ vs. $FCERT$	p-value: 1.37e-04 $\hat{A}_{12}$: 0.96	p-value: 1.39e-04 $\hat{A}_{12}$: 0.98	p-value: 6.39e-05 $\hat{A}_{12}$: 1.00
$TCERT_0$ vs. $NEZHA$	p-value: 1.66e-04 $\hat{A}_{12}$: 0.96	p-value: 1.69e-04 $\hat{A}_{12}$: 0.98	p-value: 1.71e-04 $\hat{A}_{12}$: 1.00
$TCERT_0(30K)$ vs. $FCERT(30K)$	p-value: 1.31e-04 $\hat{A}_{12}$: 0.91	p-value: 1.31e-04 $\hat{A}_{12}$: 0.91	p-value: 5.85e-05 $\hat{A}_{12}$: 0.91
$TCERT_0(30K)$ vs. $NEZHA(30K)$	p-value: 3.45e-04 $\hat{A}_{12}$: 0.88	p-value: 2.36e-03 $\hat{A}_{12}$: 0.81	p-value: 1.22e-04 $\hat{A}_{12}$: 0.91

The values are calculated using Python SciPy.

Internal validity. Threats to internal validity are mainly related to uncontrolled factors, which may also contribute to the evaluation results. In our study, the major threat is the randomness introduced to the employed algorithms. To compare these randomized algorithms, we follow the practical guidelines advised in Reference [1] for the use of statistical tests:

- We run each randomized algorithm 10 times, and not more as some algorithms take days each time;
- We report the means and standard deviations in Table 5 to help the meta-analysis of published results across studies;
- As some results shown in Table 5 (particularly *ACC* statement, *ACC* branch, and $|UDIF|$) are similar, we use the two-tailed nonparametric Mann–Whitney U-test to detect statistical differences for these metrics [39]. We conduct four groups of U-tests for pairwise comparison, namely $TCERT_0$ vs. $FCERT$, $TCERT_0$ vs. $NEZHA$, $TCERT_0(30K)$ vs. $FCERT(30K)$, and $TCERT_0(30K)$ vs. $NEZHA(30K)$. Table 9 shows that all tests are significant at the level of $\alpha = 0.005$.
- To assess the magnitude of the improvement of transcert on *ACC* and $|UDIF|$, we report Vargha and Delaney's $\hat{A}_{12}$ in Table 9, which is a nonparametric effect size measure [28, 37, 53]. The $\hat{A}_{12}$ values in Table 9 show a high probability (at least 0.8) that running transcert yields higher *ACC* and $|UDIF|$ than running frankencert and NEZHA.

We reveal that transcert introduces two types of randomness into certificate generation:

- *Randomness in seed selection*: During each iteration, transcert randomly picks up a vertex in the coverage transfer graph and then picks up a certificate corresponding to this vertex as a seed. Note that the vertex selection is not a pure-random process, since the selection needs to meet a geometric distribution (see Section 4.2.2); the vertices on the graph are dependent from each other, indicating that selection of them only relies on the distribution of their POTENTIALS.
- *Randomness in mutator selection*: transcert mutates the seeding certificate using a random mutator. The coverage transfer graph grows if the mutant explore new vertex or edge, and the POTENTIALS of each vertices are updated.

The introduced randomness without *randomness justification* may cause two side effects. (1) The coverage transfer graph cannot steadily expand. Most mutants are useless and cannot explore new paths and new vertices. (2) The coverage transfer graph obtained from each experiment varies a lot, which may cause poor effect in a single experiment due to the insufficient capability of exploring corner cases.

The randomness is justified by the coverage transfer graph. As Figure 10 shows, as the mutation process continues, the coverage transfer graph grows and gradually converges; the resulting

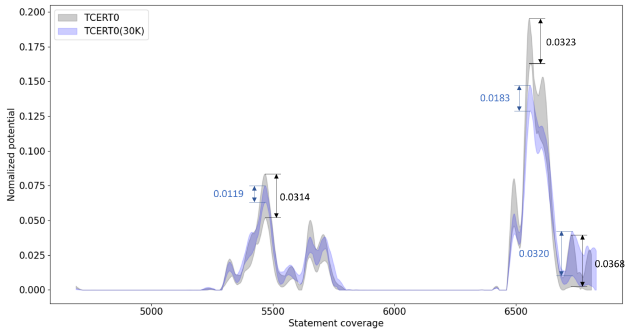


Fig. 10. Distribution of POTENTIALS of coverage transfer graph vertices. The grey and blue regions are the distribution regions of 10 $TCERT0$ graphs and 10 $TCERT0(30K)$ graphs, respectively. The x-coordinate is the statement coverage and the y-coordinate the sum of the standardized POTENTIAL of the set of vertices corresponding to that coverage.

coverage transfer graphs are similar, saying that the distributions of vertices' POTENTIALS among all experiments are similar.

We further calculate the POTENTIAL variance of the coverage transfer graphs in ten experiments. The smaller the variance, the more similar the coverage transfer graphs:

$$Variance = \sqrt{\frac{\sum_{experiment\ i} \sum_{vertex\ v} [NomalizedPotential_i(v) - AveragePotential(v)]^2}{Number\ of\ i * Number\ of\ v}}. \quad (7)$$

As a result, the variance of $TCERT_0(30K)$'s 10 graphs is 0.0001876, 51.5% smaller than that of $TCERT_0$ (0.0003869). It indicates that the coverage transfer graph converges along with the increases of iterations.

The coverage transfer graph utilizes POTENTIALS to justify the randomness in vertex selection and guides all experiments toward a similar goal—a large, complete, and convergent coverage transfer graph. However, on the basis of similar coverage transfer graphs, the corresponding test suites are very diverse. Indeed, less than 1% certificates repeats in different experiments, and neither of the two identical vertices (including certificates on them) are found in all experiments.

The resulting test suite is diversified by the randomness introduced into seed selection and mutator selection—during each iteration, a seed with different certificate components is randomly chosen, and a specific mutator is chosen for mutating the seed; the resulting certificate is different from the seed and the other certificates in the evaluation. The randomness thus significantly increases the diversity and effectiveness of the test suite.

Randomness: The randomness of transcert is introduced by random seed and mutator selection. The coverage transfer graph justifies the introduced randomness to constrained randomness, leading to a large, complete and convergent graph but a diversified test suite.

External validity. Threats to external validity mainly concern whether our results can be generalized to other situations. transcert generates the test certificates based on a set of seeds, which contains 1005 certificates prepared by the frankencert developers. However, due to the complex structure of X.509 certificate, these seeds may not be representative enough for covering all of the situations. The exploration space is constrained by the seed certificates, which poses a major threat to the effectiveness of transcert. Besides, NEZHA is re-implemented by ourselves; it differs from

the original tool in two aspects: (1) transcert's mutators, rather than those defined in LibFuzzer, are employed, and (2) it calculates δ-diversity using five SSL/TLS implementations' validation results (including error messages). Therefore, the effectiveness of NEZHA used in our study is different from that of the original tool.

6 AN ANALYSIS OF ROOT CAUSES TO VALIDATION DIFFERENCES

We analyze the validation differences revealed by the transcert certificates and classify them into four categories: CC-specific, CV-specific, CP-specific, and CRL-specific validation differences. These validation differences may come from the messy standards, the intricate validation process, and the potential flaws introduced into X.509 certificate validation code.

Messy standards. As will be explained in Section 7, certificate validation is surrounded by a number of standards. RFC documents also contain a number of soft constraints that an implementation "MAY/MAY NOT" and "SHOULD/SHOULD NOT" meet with. Developers selectively conform to these constraints when implement their implementations, leading to many validation differences among implementations.

Our test certificates also show that the primary goal of certificate chain validation is inappropriately presented in RFC 5280.

In Section 6.1, it says,

> The primary goal of path validation to verify the binding between a subject distinguished name or a subject alternative name and subject public key, as represented in the target certificate, based on the public key of the trust anchor.

SSL/TLS implementations may falsely validate bogus certificates without subject names nor subject alternative name extensions. It implies that the goal of chain validation is to verify the binding between a subject and a key, rather than that between the subject's name(s) and a key.

We reported a technical errata to the IETF PKIX group. The group responded that the inconsistent terminology and misuse of "or" and "and/or" cause misunderstandings. RFC 5280 should state

> The primary goal of path validation to verify the binding between the subject identifier(s) and the subject public key info, as represented in the target certificate, based on the public key of the trust anchor.

Intricate validation process. We identify 17 types of certificate validation differences (see Table 10), all of which have been confirmed and to the best of our knowledge, 11 of which (CC1~4, CV2, CRL1~6) have never been reported in previous studies. Furthermore, six root causes (CC1~3, CV2, CV3, and CP4) are missed by $TCERT_3$, and seven root causes (CC2~3, CV1~2, CP2~4) are missed by *NEZHA*, and another seven (CC2~4, CV1 & 3, CP1 & 3) are missed by *RCERT*, and *FCERT* only reveals two of them (CV3, CP1). These validation differences are raised in different circumstances: RFCs contain a number of constraints an implementation must/may process; the PKI structures are really complicated; implementations are flexible to conform to the RFCs.

Six CRL-specific validation differences are revealed. Few researches have ever studied CRL-specific validation differences; to the best of our knowledge, transcert is the first to use CRL mutants to differently test CRL validation.

Developers' feedback. Developers do have many doubts and concerns when implementing certificate validation code. For instance, OpenSSL security team admitted that there are a lot of

Table 10. Unique Validation Differences Revealed by transcert and Their Root Causes

	Root Cause	#UDIF	OpenSSL	mbed TLS	GnuTLS	MatrixSSL	NSS	UDIF revealed by
CC-specific	**CC1:** *Implementations can or cannot construct a certificate chain containing v1, v2, and v3 certificates.*	7	T	T	T	F	T	transcert, NEZHA, RFCcert
	CC2: *Name chaining may or may not be performed between empty names.*	1	T	F	F	F	F	transcert
	CC3: *Validators differently process self-issued certificates.*	1	T/F	T/F	T/F	T/F	T/F	transcert
	CC4: *Vaidators may or may not rely on the authority key identifier(AKI)/ subject key identifier(SKI) extensions to construct a certificate chain.*	5	F	T	F	F	T	transcert, NEZHA
CV-specific	**CV1:** *Validators differently validate a certificate whose CA bit of the basic constraints extension is True, but keyCertSign bit of the KeyUsage extension is incorrectly set.*	6	F	F	F	T	F	transcert
	CV2: *If an extension having two or more instances can be recognized, then validators differently use these instances in certificate validation.*	4	T/F	T/F	F	T/F	T/F	transcert, RFCcert
	CV3: *Validators differently accept certificates using insecure signature algorithms.*	4	T	T	F	T	F	frankencert, NEZHA
CP-specific	**CP1:** *Validators differently parse a v1 certificate having extensions.*	1	T	F	F	F	T	frankencert, transcert, NEZHA
	CP2: *Validators differently parse a certificate with more than one instances of an extension.*	3	T	F	T	T	T	transcert, RFCcert
	CP3: *Validators differently recognize critical extensions.*	7	T/F	T/F	T/F	T/F	T/F	transcert
	CP4: *Validators may accept or reject extensions with inappropriate values.*	5	T/F	F	T/F	F	F	transcert, RFCcert
CRL-specific	**CRL1:** *CRL has expired or CRL is in the future.*	2	F	F	T	/	T	transcert
	CRL2: *Non-critical extensions are marked as critical.*	1	F	F	T	/	F	transcert
	CRL3: *CRL contains invalid "IDP" extension.*	1	F	F	T	/	F	transcert
	CRL4: *The issuer of the CRL is inconsistent with the issuer of the root certificate in the certificate chain.*	2	F	T	T	/	F	transcert
	CRL5: *The revocation date of a revoked certificate is in the future.*	1	F	T	F	/	T	transcert
	CRL6: *A revoked certificate contains an invalid* `Certificate Issuer` *CRL-entry extension.*	1	T	F	F	/	F	transcert
Total		52						

Note that MatrixSSL does not support for CRL validation.

*Each root cause corresponds to one or more test certificates under validation.

*T and F denote that a validator can accept and reject the corresponding test certificates, respectively. T/F indicates a validator can accept some tests but rejects the others.

certificates that can violate RFCs, but they cannot reject all of them unless they are certainly security related.

Mbed TLS developers have confirmed a parsing error: One certificate has `RelativeDistinguishedName` that contains more than one `AttributeTypeAndValue`. They explained that they were unsure if such structured names were actually used, and thus they were hesitant to increase code complexity to cover that case. They have fixed the naming problem in their internal development branch and the patch has been included in the next release.

OpenSSL, GnuTLS, mbed TLS, and MatrixSSL behave differently when a certificate has two instances of the `SubjectKeyIdentifier`. This finding has triggered an open question on IETF

PKIX, which further requires the IETF PKIX/TLS working groups to investigate the necessity of matching AKI/SKI certificate extensions during certificate validation and inconsistencies among RFCs 4158, 5280, and 6125. ARM mbed TLS developers also started to forbid repeated extensions in X.509 certificates, because RFC 5280 states that "a certificate MUST NOT include more than one instance of a particular extension." GnuTLS developers replied that although the certificate could be accepted in certificate verification, a user would receive warning messages when printing it out.

Mbed TLS accepts certificate chains of the format $[cert_0, cert_1, cert_2]$ even if $cert_1$ has the same subject as $cert_2$, but the others (e.g., OpenSSL/GnuTLS) reject such chains and issue different warning messages (e.g., unable to get local issuer certificate, ASN signature error, or the certificate issuer is unknown). Mbed TLS developers have explained that mbed TLS accepts these certificate chains, since they have complete and valid key chains; they may reject these chains in future development.

OpenSSL, mbed TLS, and GnuTLS accept a certificate chain if the first certificate is a trusted self-signed CA certificate, but omit verifying the subsequent certificates. It does not strictly conform to the validation policy "when the trust anchor is provided in the form of a self-signed certificate, this self-signed certificate is not included as part of the prospective certification path" [18]. The developers have explained that it is useful to accept it, because many misinformed users include the trust anchor in the chain they provide, even though the RFCs recommend to reject such cases. More importantly, accepting such chains allows users to trust self-signed non-CA certificates if they choose.

SSL/TLS implementations behave differently in CRL validation. An example that triggers the "CRL5" validation difference is shown in Figure 4: mbed TLS and NSS take the CRL with future `RevocationDate` as invalid and should not revoke the end entity certificate. However, OpenSSL takes the date in the CRL as metadata and makes the CRL inactive during CRL validation.

Root causes: Our certificates reveal that the primary goal of certificate chain validation is inappropriately presented in RFC 5280. They also help identify 17 types of validation differences among SSL/TLS implementations—all of which have been confirmed and eleven have never been reported previously.

7 RELATED WORK

Next we discuss three strands of related work: (1) search-based software engineering and fuzzing approaches, (2) techniques for guaranteeing reliability of certificate validation in SSL/TLS implementations, and (3) standards surrounding X.509 certificate validation.

Search-based software engineering and fuzzing. Search-based software engineering (SBSE) proposed by Harman et al. [30, 31] casts the difficult SE problems into searching and optimization problems. Many search-based algorithms, including genetic algorithm, hill climbing, and simulated annealing, are applied to specific problems such as project management [42], requirements engineering [38], and program optimization [3, 36]. To successfully apply search-based techniques to an SE problem, engineers need to design the problem's representation, a fitness function, and manipulation operators.

Fuzzing is a popular testing technique for generating (unexpected) test inputs for testing software products [6, 7, 27]. It can be taken as an SBSE technique that searches effective test inputs for hitting software defects. Godefroid has summarized three mainstream fuzzing approaches: black-box fuzzing, grammar-based fuzzing, and white-box fuzzing [26]. Black-box fuzzing creates inputs

randomly and runs them on the objective software systems for finding security vulnerabilities, with low efficiency for generating new interesting inputs. Grammar-based fuzzing specifies the inputs' grammars to refrain invalid inputs from being generated. White-box fuzzing gathers constraints by symbolically executes the program dynamically with test inputs, and generates inputs that can cover more execution paths based on these constraints.

As a typical white-box fuzzing technique, coverage-based fuzzing, such as SAGE [8], employs heuristic search methods based on instruction coverage when exploring large space. Other fuzzers like AFL[9] and LibFuzzer[10] omit symbolic execution and constraint generation and extend the coverage-based methods to grey-box fuzzing methods. These fuzzers use heuristic methods such as simulated annealing and genetic algorithm to explore the input space.

Correspondingly, transcert is a domain-specific fuzzer that collects the code coverage for directing certificate generation. In particular, it constructs a novel coverage transfer graph that precisely abstracts the executions of the test inputs and the coverage transfers from the seeds to the mutants, and the boundary certificates are taken as seeds for certificate generation.

Guaranteeing reliability of certificate validation. Many researchers have designed techniques for guaranteeing the reliability of SSL/TLS implementations. CertShim proposed by Bates et al. [4] provides a lightweight retrofit to SSL implementations that protects against SSL vulnerabilities; it uses dynamically linked objects and binary instrumentation to implement a defense layer on SSL implementations. Beurdouche et al. have systematically tested open source TLS implementations for state machine bugs and present a verified implementation of a composite TLS state machine that can be embedded into OpenSSL [5]. Walz has proposed a response-distribution guided fuzzing algorithm for black-box testing TLS handshake [54]. Sivakorn et al. have proposed a black-box testing framework that employs automata learning algorithms for analyzing SSL/TLS hostname verification implementations [49].

Some other techniques have been proposed to test X.509 certificate validation logic, as Table 11 shows. SymCerts, proposed by Chau et al., constructs well-structured certificate chains and then leverages symbolic execution to extract constraints that may be missed checking by SSL/TLS implementations [11]. RFCcert proposed by Tian et al. extracts rules from RFCs and then employs dynamic symbolic execution to combine these rules and generate test certificates [13, 50]. Coveringcerts utilizes combinatorial testing to cover the search space [35], which is designed to exhaustively explore the search space and can be inefficient in generating some corner cases revealing validation differences.

Some fuzzing techniques leverage real-world certificates as seeds and modify them to generate new testing certificates. frankencert generates test certificates by randomly synthesizing existing certificates and then differentially tests certificate validation among SSL/TLS implementations [10]. mucert designs a series of certificate mutators and mutates existing certificates under the direction of accumulative code coverage [15]. Chen et al. have proposed DRLgencert, a deep reinforcement learning approach to fuzzing SSL/TLS implementations, which directs the certificate generation process by choosing the best next action according to the result of a previous modification [12]. NEZHA, a domain-independent framework proposed by Petsios et al., utilizes LibFuzzer[10] to modify seed certificates and employs the behavioral asymmetries between test programs to direct the generation process [43].

To the best of our knowledge, transcert is the first effort that balances the growing of a test suite and its diversity. Based on the seeds in our evaluation, transcert is more efficient in hitting validation differences than frankencert and RFCCert; during 10,000 iterations, it only misses one

[9] https://lcamtuf.coredump.cx/afl/.
[10] https://llvm.org/docs/LibFuzzer.html.

Table 11. A Comparison of Related Techniques

Name	Test object	Test approach	Need seeds	Need mutation	How to generate new certificates from seeds	How to direct the generation
HVLearn [49]	SSL/TLS hostname verification	Automata learning	\	\	\	\
Response Guided Fuzzing [54]	TLS handshake	Differential fuzz testing	\	\	\	\
SymCerts [11]	X.509 certificate validation	Symbolic execution mixed with concrete values	No	No	\	\
RFCcert [50]	X.509 certificate validation	RFC rule extraction & dynamic symbolic execution	No	No	\	\
Coveringcerts [35]	X.509 certificate validation	Combinational testing	No	No	\	\
frankencert [10]	X.509 certificate validation	Differential fuzz testing	Yes	No	Combining fragments of real-world certificates	Random
mucert [15]	X.509 certificate validation	Differential fuzz testing	Yes	Yes	Defining mutators to modify the seeding certificates	Accumulate code coverage
DRLgencert [12]	X.509 certificate validation	Differential fuzz testing	Yes	Yes	Defining actions for modifying the seeding certificates	Deep reinforcement learning
NEZHA [43]	X.509 certificate validation	Differential fuzz testing	Yes	Yes	Using LibFuzzer to modify the seeding certificates	Behavioral asymmetries (δ-diversity)
transcert (ours)	X.509 certificate validation	Differential fuzz testing	Yes	Yes	Defining certificate and CRL mutators.	A coverage transfer graph

validation difference reported by frankencert and another reported in Reference [35] and finds many new differences missed by these techniques.

Standards surrounding certificate validation. Certificate validation is specified in a number of standards, including RFCs 2246 [20], 2527 [16], 3647 [17], 4346 [21], 5246 [22], 5280 [18], 5878 [9], 6101 [24], 6125 [45], 6818 [55], 8398 [41], 8399 [32], 8446 [44], and ITU-T X.509 | ISO/IEC 9594-8 [34].

Among them, the role of certificate validation is usually specified in the SSL/TLS protocols. The SSL protocol do have two publicly released versions: SSLv2 released in 1995 [23] and SSLv3 in 2011 [24]. The SSL protocol was replaced by TLS 1.0 [20] in 1999 and subsequently TLS 1.1 in 2002 [21], 1.2 in 2006 [22], and 1.3 in 2018 [44]. Note that although SSLv2 and SSLv3 have been deprecated by RFCs 6176 [52] and 7568 [2], many implementations have permitted the negotiation of SSLv2 and SSLv3.

The certificate validation process and the associated constraints are also specified in several standards. ISO/IEC 9594-8/ITU-T X.509 specifies the X.509 certificate structure [34]. RFCs 3280 [33], 3709 [48], 4325 [47], and 5280 [18] provide the Internet X.509 PKI certificate and CRL profile. RFCs 4158 [19] and 5280 [18] specify how a certification path is built and how a certificate chain is validated, respectively. RFC 7773 provides a mechanism allowing an application to obtain information that expresses the identity of a subject in an X.509 certificate [46]. Many RFCs are also proposed for specifying domain-specific certificates and their extensions (e.g., the X.509 Attribute certificate in RFC 5878 [9] and the NSA certificates specified in RFC 7169 [51]).

Obviously, it is not easy for the above specifications to be consistent with each other; it is also not possible for developers to follow only one or two RFCs in their development. The messy standards and the inconsistencies among them become one main source of validation differences.

8 CONCLUSION

Certificate validation is critical for establishing secure connections over the Internet. We propose transcert—a directed approach to testing X.509 certificate validation via coverage transfer graphs. The evaluation results clearly show the effectiveness of transcert: It can continuously explore the search space and is more effective in exposing a variety of validation differences than frankencert, RFCcert, and NEZHA; it also supplements RFCcert in conformance testing of the SSL/TLS implementations against validation rules. We believe that transcert provides a practical solution to enhancing the reliability of X.509 certificate validation and the robustness of SSL/TLS implementations. transcert is publicly available at https://github.com/yuemonangong/transcert.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We also thank SSL/TLS developers for analyzing our reported issues. This submission is an extension to our early work in ICSME 2020 and ESEC/FSE 2015.

REFERENCES

[1] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verif. Reliabil.* 24, 3 (2014), 219–250.

[2] Richard L. Barnes, Martin Thomson, Alfredo Pironti, and Adam Langley. 2015. Deprecating secure sockets layer version 3.0. *RFC* 7568 (2015), 1–7.

[3] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. 2017. Optimising darwinian data structures on google guava. In *Search Based Software Engineering*, Tim Menzies and Justyna Petke (Eds.). Springer International Publishing, Cham, 161–167.

[4] Adam Bates, Joe Pletcher, Tyler Nichols, Braden Hollembaek, Dave Tian, Kevin R. B. Butler, and Abdulrahman Alkhelaifi. 2014. Securing SSL certificate verification through dynamic linking. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 394–405.

[5] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A messy state of the union: Taming the composite state machines of TLS. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'15)*. IEEE Computer Society, 535–552.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1032–1043.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Softw. Eng.* 45, 5 (2019), 489–506.

[8] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 122–131. https://doi.org/10.1109/ICSE.2013.6606558

[9] Mark Brown and Russell Housley. 2010. Transport layer security (TLS) authorization extensions. *RFC* 5878 (2010), 1–19.

[10] 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 *Proceedings of the IEEE Symposium on Security and Privacy (SP'14)*. IEEE Computer Society, 114–129.

[11] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. SymCerts: Practical symbolic execution for exposing noncompliance in X.509 certificate validation implementations. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'17)*. IEEE Computer Society, 503–520.

[12] 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 *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*. IEEE Computer Society, 48–58.

[13] Chu Chen, Cong Tian, Zhenhua Duan, and Liang Zhao. 2018. RFC-directed differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, 859–870.

[14] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. IEEE/ACM, 1257–1268.

[15] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. Association for Computing Machinery, New York, NY, 793–804. https://doi.org/10.1145/2786805.2786835
[16] Santosh Chokhani and Warwick Ford. 1999. Internet X.509 public key infrastructure certificate policy and certification practices framework. *RFC* 2527 (1999), 1–45.
[17] Santosh Chokhani, Warwick Ford, Randy Sabett, Charles R. Merrill, and Stephen S. Wu. 2003. Internet X.509 public key infrastructure certificate policy and certification practices framework. *RFC* 3647 (2003), 1–94.
[18] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and W. Timothy Polk. 2008. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. *RFC* 5280 (2008), 1–151.
[19] Matt Cooper, Yuriy Dzambasow, Peter Hesse, Susan Joseph, and Richard Nicholas. 2005. Internet X.509 public key infrastructure: Certification path building. *RFC* 4158 (2005), 1–81.
[20] Tim Dierks and Christopher Allen. 1999. The TLS protocol version 1.0. *RFC* 2246 (1999), 1–80.
[21] Tim Dierks and Eric Rescorla. 2006. The transport layer security (TLS) protocol version 1.1. *RFC* 4346 (2006), 1–87.
[22] Tim Dierks and Eric Rescorla. 2008. The transport layer security (TLS) protocol version 1.2. *RFC* 5246 (2008), 1–104.
[23] Dr. Taher Elgamal and Kipp E. B. Hickman. 1995. *The SSL Protocol*. Internet-Draft draft-hickman-netscape-ssl-00. Internet Engineering Task Force (unpublished).
[24] Alan O. Freier, Philip Karlton, and Paul C. Kocher. 2011. The secure sockets layer (SSL) protocol version 3.0. *RFC* 6101 (2011), 1–67.
[25] 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 (CCS'12)*. Association for Computing Machinery, New York, NY, 38–49. https://doi.org/10.1145/2382196.2382204
[26] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70–76.
[27] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'08)*, Vol. 8. The Internet Society, 151–166.
[28] Keith J. Goulden. 2006. Effect sizes for research: A broad practical approach.
[29] Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. IEEE, 621–631.
[30] Mark Harman. 2007. The current state and future of search based software engineering. In *Proceedings of the Annual Conference on the Future of Software Engineering (FOSE'07)*. 342–357. https://doi.org/10.1109/FOSE.2007.29
[31] Mark Harman and Bryan F. Jones. 2001. Search-based software engineering. *Inf. Softw. Technol.* 43, 14 (2001), 833–839.
[32] Russ Housley. 2018. Internationalization updates to RFC 5280. *RFC* 8399 (2018), 1–9.
[33] Russell Housley, W. Timothy Polk, Warwick Ford, and David Solo. 2002. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. *RFC* 3280 (2002), 1–129.
[34] International Telecommunication Union. 2019. Recommendation ITU-T X.509 (Open Systems Interconnection: The Directory: Public-key and attribute certificate frameworks).
[35] Kristoffer Kleine and Dimitris E. Simos. 2017. Coveringcerts: Combinatorial methods for X.509 certificate testing. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST'17)*. IEEE Computer Society, 69–79.
[36] William B. Langdon and Mark Harman. 2014. Optimizing existing software with genetic programming. *IEEE Trans. Evol. Comput.* 19, 1 (2014), 118–135.
[37] Nancy L. Leech and Anthony J. Onwuegbuzie. 2002. A call for greater use of nonparametric statistics. (2002).
[38] Lingbo Li, Mark Harman, Fan Wu, and Yuanyuan Zhang. 2016. The value of exact analysis in requirements selection. *IEEE Trans. Softw. Eng.* 43, 6 (2016), 580–596.
[39] H. B. Mann and D. R. Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.* 18, 1 (1947), 50–60.
[40] William M. McKeeman. 1998. Differential testing for software. *Dig. Techn. J.* 10, 1 (1998), 100–107.
[41] Alexey Melnikov and Weihaw Chuang. 2018. Internationalized email addresses in X.509 certificates. *RFC* 8398 (2018), 1–12.
[42] Leandro L. Minku, Dirk Sudholt, and Xin Yao. 2012. Evolutionary algorithms for the project scheduling problem: Runtime analysis and improved design. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'12)*. Association for Computing Machinery, New York, NY, 1221–1228. https://doi.org/10.1145/2330163.2330332
[43] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient domain-independent differential testing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'17)*. IEEE Computer Society, 615–632.
[44] Eric Rescorla. 2018. The transport layer security (TLS) protocol version 1.3. *RFC* 8446 (2018), 1–160.

[45] 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.
[46] Stefan Santesson. 2016. Authentication context certificate extension. *RFC* 7773 (2016), 1–16.
[47] Stefan Santesson and Russell Housley. 2005. Internet X.509 public key infrastructure authority information access certificate revocation list (CRL) extension. *RFC* 4325 (2005), 1–7.
[48] Stefan Santesson, Russ Housley, and Trevor Freeman. 2004. Internet X.509 public key infrastructure: Logotypes in X.509 certificates. *RFC* 3709 (2004), 1–21.
[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 *Proceedings of the IEEE Symposium on Security and Privacy (SP'17)*. IEEE Computer Society, 521–538.
[50] Cong Tian, Chu Chen, Zhenhua Duan, and Liang Zhao. 2019. Differential testing of certificate validation in SSL/TLS implementations: An RFC-guided approach. *ACM Trans. Softw. Eng. Methodol.* 28, 4 (2019), 24:1–24:37.
[51] Sean Turner. 2014. The NSA (No Secrecy Afforded) certificate extension. *RFC* 7169 (2014), 1–3.
[52] Sean Turner and Tim Polk. 2011. Prohibiting secure sockets layer (SSL) version 2.0. *RFC* 6176 (2011), 1–4.
[53] András Vargha and Harold D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *J. Educ. Behav. Stat.* 25, 2 (2000), 101–132.
[54] Andreas Walz and Axel Sikora. 2018. Maximizing and leveraging behavioral discrepancies in TLS implementations using response-guided differential fuzzing. In *Proceedings of the International Carnahan Conference on Security Technology (ICCST'18)*. IEEE, 1–5.
[55] Peter E. Yee. 2013. Updates to the internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. *RFC* 6818 (2013), 1–8.

Received 23 January 2021; revised 26 November 2021; accepted 3 January 2022