Guided Differential Testing of Certificate Validation in SSL/TLS Implementations

Yuting Chen
Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
chenyt@cs.sjtu.edu.cn

Zhendong Su
Department of Computer Science
University of California, Davis, USA
su@cs.ucdavis.edu

ABSTRACT

Certificate validation in SSL/TLS implementations is critical for Internet security. There is recent strong effort, namely *frankencert*, in automatically synthesizing certificates for stress-testing certificate validation. Despite its early promise, it remains a significant challenge to generate effective test certificates as they are structurally complex with intricate syntactic and semantic constraints.

This paper tackles this challenge by introducing *mucert*, a novel, guided 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) diversify them by adapting Markov Chain Monte Carlo (MCMC) sampling. The diversified certificates are then used to reveal discrepancies, thus potential flaws, among different certificate validation implementations.

We have implemented mucert and extensively evaluated it against frankencert. Our experimental results show that mucert is significantly more cost-effective than frankencert. Indeed, $1K$ *mucerts* (*i.e.*, mucert-mutated certificates) yield *three times* as many *distinct* discrepancies as $8M$ *frankencerts* (*i.e.*, frankencert-synthesized certificates), and 200 mucerts can achieve higher code coverage than $100,000$ frankencerts. This improvement is significant as it incurs much cost to test each generated certificate. We have analyzed and reported *20+ latent discrepancies* (presumably missed by frankencert), and reported an additional *357 discrepancy-triggering certificates* to SSL/TLS developers, who have already confirmed some of our reported issues and are investigating causes of all the reported discrepancies. In particular, our reports have led to bug fixes, active discussions in the community, and proposed changes to relevant IETF's RFCs. We believe that mucert is practical and effective for helping improve the robustness of SSL/TLS implementations.

More information on mucert and our results can be found at `http://stap.sjtu.edu.cn/~chenyt/mucert.html`.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing)*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786835

General Terms

Algorithms

Keywords

Differential testing, mutation, certificate validation

1. INTRODUCTION

Secure Sockets Layer (SSL) [27] and Transport Layer Security (TLS) [24] are cryptographic protocols for security protection over the Internet. Various implementations and libraries (*e.g.*, OpenSSL [1] and NSS [2]) exist to support the protocols; they facilitate the incorporation of SSL/TLS in user applications. All web browsers also support users in establishing SSL/TLS connections.

X.509 certificates provide the principal medium for websites and Internet users to authenticate each other and establish secure SSL/TLS connections. For example, when a web browser requests an `https` connection to a website, it will retrieve the site's X.509 certificate and validate it. If the certificate fails in validation, the browser will display a warning message to the user, who may then refuse this connection. But similar to any real-world software, SSL/TLS implementations or libraries may contain defects, and in particular may not validate X.509 certificates correctly, making certification validation the most dangerous code in the world [28]. Indeed, certificate validation has been completely broken in many security-critical applications and libraries [28, 31]; defects can be embedded into certificate validation code, making SSL/TLS connections completely vulnerable or insecure.

Certificate validation mainly checks, given a server certificate, whether it is well formed, whether it has not expired, and whether it is issued by a trusted certificate authority (CA). However, all SSL/TLS implementations validate X.509 certificates by following a complicated, ad-hoc process described in several RFCs (including RFC 2246, 2527, 2818, 4346, 5246, 5280, 6101, 6125) [19, 21–24, 27, 40, 41]. Developers must define their respective validation policies for handling ambiguous descriptions (*e.g.*, "*the serial number MUST be a positive integer assigned by the CA to each certificate ... non-conforming CAs may issue certificates with serial numbers that are negative or zero. Certificate users SHOULD be prepared to gracefully handle such certificates*" [21]). Developers can also make minor mistakes, such as misunderstanding the SSL/TLS application program interfaces (APIs), using insecure middleware or libraries, and breaking/disabling certificate validation [28].

Furthermore, existing SSL/TLS implementations are not adequately tested before being released, as test certificates have to be designed elaborately and mainly manually. One main reason is that X.509 certificates are themselves structurally complex data: each certificate is composed of several fields for identifying itself and a sequence of extensions, each field can encompass semantic and syntactic constraints, and certificates must be carefully organized into certificate chains. Thus, any test certificate must be constructed to conform to or deliberately violate the constraints, which makes manual testing inevitably inadequate.

Brubaker *et al.* [13] propose the first automated technique, called *frankencert*, to randomly combine 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 and tested, so it is very resource-intensive, but most of the frankencerts do not trigger any discrepancies. In particular, 8,127,600 frankencerts could only yield 208 discrepancies, which further reduce to only 9 *distinct* ones, among the many SSL/TLS implementations [13].

Inspired by Brubaker *et al.*'s work and recognizing its limitation, we aim to generate effective test certificates, and in particular our goal is to generate "diverse" certificates for testing. 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. We cast effective test certificate generation as an optimization problem: *Given certificates* $Cert = \{cert_0, cert_1, ..., cert_n\}$, *construct* $Cert' = \{cert'_0, cert'_1, ..., cert'_n\}$ *whose certificates are as diverse as possible.*

To this end, we introduce *mucert*, a novel, guided approach to differential testing of certificate validation. In particular, mucert adapts Markov Chain Monte Carlo (MCMC) sampling to diversify certificates. MCMC methods are a class of algorithms for sampling from a probability distribution by constructing a Markov chain that converges to the desired distribution [18, 36]. For many intractable problems without exact optimization algorithms, MCMC sampling provides a general solution [42]. Section 2 discusses detailed design and technical challenges that we tackle to realize mucert, such as choosing the acceptance condition and mutation operations.

This paper makes the following main contributions:

- *Problem formulation.* We cast the difficult problem of certificate generation as an optimization problem, which allows us to leverage the many easily accessable Internet certificates and transform them to effectively test certificate validation logic. This high-level view is general and may be applicable in other settings with structurally complex test inputs.
- *MCMC-guided certificate mutation.* We adapt MCMC sampling to effectively diversify certificates. In particular, we use code coverage to guide the sampling process to accept and retain representative certificates in the test suite. To our knowledge, this work is the first to utilize MCMC sampling for generating diverse test inputs in differential testing.
- *Implementation and evaluation.* We have implemented mucert and compared it against frankencert and two other mutation algorithms on 9 real-world SSL/TLS implementations. Our results show that mucert significantly outperforms frankencert and the other techniques. Most notably, $1K$ mucerts lead to 3× as many distinct discrepancies as $8M$ frankencerts, demonstrating that mucert effectively diversifies certificates.
- *Community feedback and impact.* We have also reported 20+ issues and an additional 357 discrepancy-triggering certificates, and have already received confirmations and positive feedback from the SSL/TLS developers. For example, as a result of our reports, ARM mbed TLS-1.3.10 (formerly known as PolarSSL) started to forbid repeated extensions in X.509 certificates, and active discussions on the reported certificates have led to proposed changes to IETF's relevant RFCs.

The rest of the paper is structured as follows. Section 2 presents the details of our guided technique, including basic background on certificate validation, MCMC-guided certificate diversification, and the differential testing process. We next describe our extensive evaluation of mucert against *frankencert* and two other certificate mutation algorithms to demonstrate mucert's effectiveness (Section 3). Section 4 surveys related work, and Section 5 concludes.

2. APPROACH

This section presents the technical details of our approach. We discuss necessary background on certification validation, introduce our MCMC-guided certificate mutation algorithm, and describe our differential testing process.

2.1 Background: X.509 Certificate Validation

An input to certificate validation is a chain of X.509 certificates. Each certificate consists of a sequence of three required fields [21]:

- *Certificate*, which contains a subject and an issuer, a public key associated with the subject, a validity period, and other information. An X.509 v3 certificate also has extensions that can convey such data as additional subject identification information, policy information, and certification path constraints;
- *Certificate Signature Algorithm*, the identifier for the signature algorithm used by a certificate authority (CA) to sign this certificate; and
- *Certificate Signature*, a digital signature for the certificate.

In a public key infrastructure (PKI), a certificate does not exist in isolation, but is recursively organized, together with its issuers, into a *certificate chain*. A certificate chain usually starts with an end-entity certificate followed by a list of certificates and the CA certificates. Figure 1 illustrates a typical certificate chain, where the issuer of each certificate is the subject of the next certificate, each certificate is signed by the next certificate, and the last certificate is a self-signed trust anchor. Given a certificate, an SSL/TLS implementation mainly validates whether it can be chained to a "trusted root" certificate and whether each certificate on the chain is valid at the current time [21].

2.2 Guided Certificate Optimization

Intuitively, one may collect X.509 certificates from the Internet to test SSL/TLS implementations, although these real certificates unlikely expose flaws in validation code. On the other extreme is frankencert [13], which randomly combines pieces of real certificates to construct fake ones, most of which are invalid and useless for testing. Instead, we strive to find a sweet spot between the two extremes by systematically and continuously mutating a set of real certificates to make them diverse (*i.e.*, following different program paths, triggering different validation policies, or triggering different error handlers) for testing certificate validation logic.

2.2.1 MCMC Sampling and Fitness Function

For our purpose, we adapt MCMC sampling to optimize a test suite with a fixed number of test certificates. We utilize code coverage as the fitness function to balance our optimization goal and the easiness of measurement. Another reason for using code coverage as the fitness function is that code coverage is shown to strongly correlate with the output uniqueness of a test suite, one manifestation of its diversity [10].

Let $Cov(Cert)$ be defined for computing the code coverage of a test suite $Cert$ *w.r.t.* a given SSL/TLS implementation (*e.g.*, OpenSSL):

$$Cov(Cert) \quad = \quad Cov(cert_0) \oplus \cdots \oplus Cov(cert_n),$$

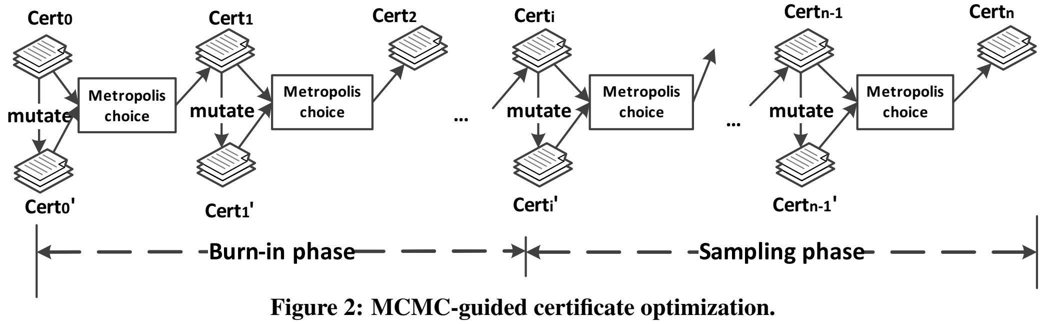


Figure 2: MCMC-guided certificate optimization.

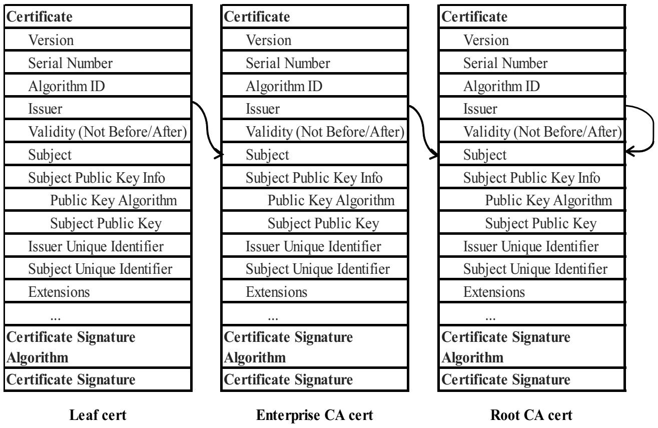


Figure 1: A typical certificate chain.

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

In our setting, code coverage helps make the test suite accept "fresh" test certificates. Let $cert$ in $Cert$ be mutated to $cert'$ in $Cert'$ (see Section 2.2.3). Let $Cov(Cert) < Cov(Cert')$. The certificate $cert'$ is obviously distinct from $cert$ or any other certificates in $Cert'$, as it exploits some new validation code (or branches). Therefore, code coverage provides the first means to approach the optimization goal, that is, given the test suite $Cert$, how can it be mutated continuously to $OptimizedCert$ such that

$$Cov(Cert) \ll Cov(OptimizedCert) \leq \top,$$

where $Cov(Cert) \ll Cov(OptimizedCert)$ denotes that the coverage should be increased as much as possible, and $\top$ is an upper bound of coverage that can be achieved by any test suite.

MCMC sampling provides another opportunity, even if the coverage of a test suite cannot get increased, to stochastically diversify the test certificates inside. MCMC advocates the idea of sampling from a probability distribution by constructing a Markov chain which converges to the desired distribution. When applied to optimization, MCMC sampling can work as an intelligent hill climbing method, and thus creates sufficient samples most of which will be taken from the optimal values of the adopted fitness function. In our setting, each sample corresponds to a test suite.

2.2.2 Sampling Process

Algorithm 1 MCMC-guided algorithm to optimize certificates

```
Input: certificate corpus $CertStore$, $n$, $k$
Output: test suite of $n$ certificates
Select $n$ random certificates from $CertStore$ and add to $Cert$
$highest\_cov \leftarrow Cov(Cert)$
$OptimizedCerts \leftarrow \{Cert\}$
repeat
    $cert \leftarrow random.choice(Cert)$
    $mutator \leftarrow random.choice(Mutator)$
    $cert' \leftarrow apply(mutator, cert)$
    $Cert' \leftarrow (Cert \backslash \{cert\}) \cup \{cert'\}$
    if $highest\_cov < Cov(Cert')$ then
        $highest\_cov \leftarrow Cov(Cert')$
        $OptimizedCerts \leftarrow \{Cert'\}$
    else if $highest\_cov == Cov(Cert')$ then
        $OptimizedCerts \leftarrow OptimizedCerts \cup \{Cert'\}$
    Accept $Certs'$ according to $A(Cert \rightarrow Cert')$
    if accepted then
        $Cert \leftarrow Cert'$
until $highest\_cov$ has not been increased for $k$ steps
$OptimizedCert \leftarrow random.choice(OptimizedCerts)$
return $OptimizedCert$
```

For MCMC sampling, mucert at first transforms the fitness function into a probability density function, by following a commonly used approach [42]:

$$P(Cert) = \frac{1}{Z}\exp(-\beta(Cov(Cert) - \perp)),$$

where β is a constant, Z a partition function that normalizes the distribution, and $\perp$ a lower bound of coverage that can be achieved by a test suite.

Mucert then adopts the Metropolis-Hastings algorithm [36] for generating Markov chains. The Metropolis-Hastings algorithm is an MCMC method for obtaining random samples from a probability distribution. It works by generating a sequence of samples whose distribution closely approximates the desired distribution; samples are produced iteratively, with the distribution of the next sample (say s') being dependent only on the current one (say s). As Figure 2 shows, in the burn-in phase, the coverage of the samples increases rapidly, while in the sampling phase, all samples hold high coverage values, but their diversities stochastically vary. We use Metropolis choice $A(s \rightarrow s')$ for sampling acceptance or rejection:

$$A(s \rightarrow s') = min(1, \frac{P(s')}{P(s)} \cdot \frac{g(s' \rightarrow s)}{g(s \rightarrow s')})$$

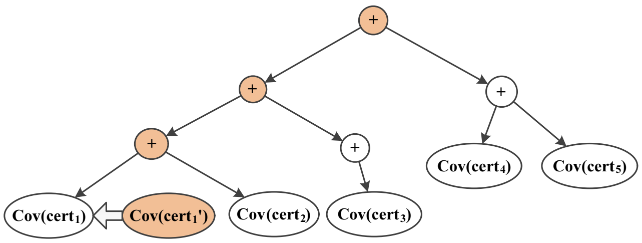


Figure 3: An example of coverage information tree. When one leaf node ($Cov(cert_1)$) is updated, the tree is updated in a bottom up style. All updated nodes are colored in brown.

where $g(s \to s')$ is the proposal distribution describing the conditional probability of proposing a new sample s' given s.

The proposal distribution in our setting is symmetric, thus the acceptance probability is reduced to

$$\begin{aligned} A(Cert \to Cert') &= min(1, \frac{P(Cert')}{P(Cert)}) \\ &= min(1, \exp(\beta(cov_1 - cov_2))), \end{aligned}$$

where $cov_1 = Cov(Cert)$ and $cov_2 = Cov(Cert')$. The acceptance probability can be directly computed from the coverage function $Cov(\cdot)$.

Let β be a negative constant in $(-1.0, 0)$. The importance of $A(Cert \to Cert')$ is: if $Cov(Cert) \leq Cov(Cert')$, the proposal is always accepted; otherwise the proposal is accepted with a certain (small) probability (namely $\exp(-\beta(Cov(Cert') - Cov(Cert)))$). Further, the smaller $Cov(Cert') - Cov(Cert)$ is, the less the acceptance probability will be. For example, let β be -0.033, $Cov(Cert)$ be 7200 (SLOC); let $Cov(Cert_1)$ and $Cov(Cert_2)$ be 7180 (SLOC) and 7170 (SLOC), respectively. $Cert_1$ is easier to accept than $Cert_2$ because

$$\begin{aligned} A(Cert \to Certs_1) &= 0.517, \text{ and} \\ A(Cert \to Certs_2) &= 0.371. \end{aligned}$$

Algorithm 1 shows the mucert algorithm for certificate optimization. It first selects a test suite of n certificates (certificate chains more rigorously). It then performs a number of iterations. During each iteration, exactly one certificate is chosen and mutated. The algorithm chooses one sample with the highest coverage as an optimal solution (theoretically any sample can be chosen for testing). Notice that mucert mutates an X.509 certificate (or a certificate chain) by rewriting the certificate (or rewriting one certificate in the chain), which we will explain in Section 2.2.3.

2.2.3 *Certificate Mutation*

We define 37 mutators (mutation operations) for supporting certificate mutation. Mucert randomly picks a certificate (or a chain) and mutates it, expecting that the mutant can exploit some new validation policies in the validation code. As Table 1 shows, these mutators are classified into two categories:

1. *Chain mutator.* A chain 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, so that each certificate is issued by the subsequent one.
2. *Certificate mutator.* A certificate mutator is used to update a single certificate, *e.g.*, rewriting the expiration date or adding an extension to the certificate. Nevertheless, 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. Further, we can mutate a certificate by deleting one of its fields in order to check whether the mutant can trigger some parsing errors or validation problems.

Table 1: Sample mutation operations used.

Category	Operations
Chain mutator	(1) Insert a certificate at a given position of a chain (2) Append a certificate to a chain (3) Delete a certificate from a chain (4) Replace one certificate in a chain with another
Certificate mutator	(5) Rewrite a certificate field (*e.g.*, notAfter, notBefore, serial number, subject, extensions) (6) Rewrite a subject field of a certificate (*e.g.*, countryname, stateOrProvinceName, stateOrProvinceName, localityName, organizationname, organizationalUnitName, commonName, emailAddress, name, title) (7) Append a set of extensions to the certificate (8) Append one extension to the certificate (9) Rewrite the criticality of one extension (10) Rewrite one extension (11) Delete a certificate field or a subject filed

We prefer a grafting strategy when rewriting a certificate or its field. Given a certificate chain, we can replace one certificate with an "invader" certificate that is randomly chosen from the certificate corpus. Similarly, we can insert the invader into the chain, or use its field to update the corresponding field of a certificate in the chain. We can also choose one or more extensions of the invader certificate and add them into a certificate. Such a strategy helps produce syntactically correct mutants that otherwise might have been rejected early during validation due to trivial parsing errors.

2.3 Differential Testing

We have implemented a testing framework in Python to realize and utilize mucert. Figure 4 illustrates the framework, which contains two key components: (1) test certificate optimization, and (2) differential testing:

- *Certificate optimization.* Mucert selects a set of n certificates at random and then performs an MCMC sampling process. One sample with the highest coverage is chosen for testing.
- *Differential testing.* Differential testing is a mature testing technology for large software systems [30, 35]: a test case is randomly generated, and output is compared for a variety of systems. In our work, we employ the generated mucerts to test commonly used SSL/TLS implementations (including OpenSSL [1], PolarSSL [3], Gnutls [4], NSS [2], CyaSSL [5], and MatrixSSL [6]) and web browsers (including Google's Chrome [7], Mozilla's Firefox [8], and Microsoft's Internet Explorer [9]) and then compare the validation results. Any behavior discrepancies among these implementations become oracles for finding flaws in their certificate validation code.

2.3.1 *Certificate Validation in Testing*

An SSL/TLS implementation can validate an X.509 certificate (or a certificate chain) in either a file mode or a client-server (C-S) mode, or both. The file mode provides a rather simple validation

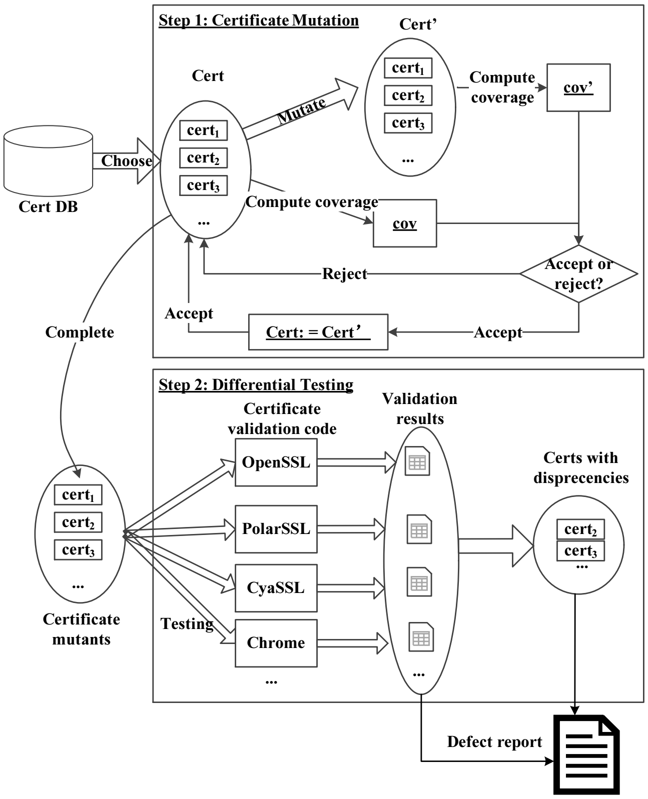


Figure 4: Guided differential testing of certificate validation.

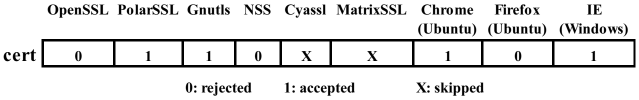
style, allowing the implementation to load and validate a PEM file containing one or more certificates. The C-S mode provides a typical, but slightly more complicated validation solution, requiring a client to retrieve a server certificate and then validate it. Table 2 shows the supportability of the SSL/TLS tools and browsers to the two validation modes.

Note that CyaSSL and MatrixSSL do not provide released utilities for validating certificate files. Although the C-S validation mode is supported by all of the implementations, it is still difficult to validate a large number of certificates since each website is usually secured by only one certificate. A browser mainly validates a certificate when it connects to a server, while up to now, we do not have any tools that can forge a web browser when it connects to a spoofed domain name (matching to the domain name appearing in a certificate). Thus mucert adopts both modes to test SSL/TLS implementations:

1. Test OpenSSL, PolarSSL, Gnutls, and NSS in file mode;
2. Test Cyassl and MatrixSSL in C-S mode. We use the server in Brubaker *et al.* [13] to warp and send the mutated certificates, and use the clients released in CyaSSL/MatrixSSL to retrieve and validate the certificates. Each client takes three arguments (host, port, path to the file with trusted root certificates) and makes an SSL 3.0 connection to the host/port. The client records the validation results, including error codes if any;
3. Import the certificates into the certificate databases used by the web browsers (Chrome, Firefox, and Internet Explorer). We assume that a validation is performed when a certificate is imported[1]. We also use the browsers to connect a localhost server on which fake certificates with the common name "`localhost`" are deployed and check whether the browsers can be forged in SSL/TLS connections.

Table 2: Supported validation modes by SSL/TLS implementations.

SSL/TLS tools or libraries	File mode (with standalone validation utility or certificate manager)	C-S mode
OpenSSL 1.0.1j	Y (openssl)	Y
PolarSSL 1.3.9	Y (cert_app)	Y
Gnutls 3.3.10	Y (certtool)	Y
NSS 3.17.3	Y (certutil)	Y
CyaSSL 3.3.0	/	Y
MatrixSSL 3.7-1	/	Y
Chrome 39.0.2171.95	Y (an OS-level certificate manager)	Y
Mozilla's Firefox 34.0	Y (PSM/NSS)	Y
Internet Explorer 11.0.14	Y (Microsoft Management Console)	Y



	OpenSSL	PolarSSL	Gnutls	NSS	Cyassl	MatrixSSL	Chrome (Ubuntu)	Firefox (Ubuntu)	IE (Windows)
cert	0	1	1	0	X	X	1	0	1

0: rejected 1: accepted X: skipped

Figure 5: Result encoding example.

2.3.2 Discrepancy Representation

The validation results among the SSL/TLS implementations can be discrepant, indicating that a certificate can be accepted by some implementations, but rejected by the others. In this work, we encode the validation results for facilitating computation of the diversity of a test suite and identification of the subtle discrepancies and their root causes. Let each validation result be simplified to "rejected" (0) or "accepted" (1). As Figure 5 illustrates, the validation results for a certificate can be encoded into a sequence of bits, representing that the certificate is accepted by PolarSSL/GNUtls/Chrome/IE, but rejected by OpenSSL/NSS/Firefox. In cases where the validation is skipped (*e.g.*, due to a server connection error), we mark the corresponding bit as "X". Therefore, a behavior *discrepancy* can appear if a sequence is not all zeros or all ones ("X" is omitted); two discrepancies can be classified into one category if their encoded results are equal (again "X" is omitted). Theoretically, a sequence of k bits has at most 2^k possible values. In our setting, the results can be reduced to at most 510 $(= 2^9 - 2)$ distinct discrepancies.

3. EMPIRICAL EVALUATION

We have conducted an extensive evaluation to compare mucert with frankencert and two other mutation algorithms. Our results show that mucert is significantly more cost-effective than the other algorithms. In particular, 200 mucerts can achieve higher code coverage than $100,000$ frankencerts. More importantly, mucert diversifies a test suite even if its coverage is not increased, making $1K$ mucerts yield *2.2 times* as many *distinct* discrepancies as $100,000$ frankencerts in the experiment and *3 times* as many as those reported by Brubaker *et al.* [13].

We have reported 20+ latent discrepancies and an additional 357 discrepancy-triggering certificates to SSL/TLS developers, who have been investigating causes of the reported discrepancies and identifying flaws in their implementations. Our reported certificates have also led to active discussions in the community and even proposed changes to IETF's RFCs. The rest of the section presents our detailed results and analysis.

[1] In fact, a certificate manager does not validate the trusted certificates (see Section 3.4).

3.1 Setup

Our empirical evaluation of mucert is designed to answer the following research questions:

- **Coverage**: What coverage can be achieved by mucerts when used for testing of SSL/TLS implementations?
- **Precision**: How precise are the mucerts for uncovering discrepancies among certificate validation code?
- **Diversity**: Are the mucerts diverse?
- **Flaws**: Can the discrepancies pinpoint any real flaws in certificate validation code?

3.1.1 Preparation

Mucert optimizes a set of certificates iteratively, guided by achieved code coverage *w.r.t.* a specific SSL/TLS implementation. For the initial seed certifates, we use a collection of 1,006 certificates provided by frankencert [13]. These certificates were gathered using ZMap [25] by scanning the Internet and attempting SSL connections to hosts listening on port 443. If a connection was successful, the certificate presented by the server was added to the collection.

As for the reference SSL/TLS implementation, we use OpenSSL-1.0.1j, and the objective is to optimize mucerts to cover the source code of OpenSSL as much as possible. We consider both *statement-* and *branch-*coverage optimized search heuristics (which we call SOSH and BOSH repsectively) to direct certificate mutations. We use the mature, widely adopted coverage tool GCOV + LCOV to collect coverage statistics.

Our evaluation was conducted on a 64-bit Ubuntu 14.04 LTS desktop (with an Intel Core i7-4770 CPU and 16GB RAM). We performed differential testing on nine SSL tools and browsers, including OpenSSL 1.0.1j, PolarSSL 1.3.9, Gnutls 3.3.10, NSS 3.17.3, CyaSSL-3.3.0, MatrixSSL 3.7-1, Google's Chrome 39.0.2171.95, Mozilla's Firefox 34.0, and Microsoft's Internet Explorer 11.0.14. Except Internet Explorer, all these tools and browsers were tested on the 64bit Ubuntu 14.04 LTS machine. Internet Explorer was tested on a 32-bit Windows 7 Enterprise desktop (with an Intel Core i5-2430M CPU and 4GB RAM).

3.1.2 Evaluated Methods and Metrics

We evaluated mucert against three other methods:

- *frankencert*: Frankencert produces a number of fake certificates that are randomly synthesized from parts of real certificates. Thus frankencerts include unusual combinations of extensions and constraints;
- *randmut*: It is a random mutation algorithm that we designed to compare against mucert. It performs random mutation operations on the certificates in a test suite; and
- *greedymut*: It is a greedy mutation algorithm that we designed also for the purpose of demonstrating mucert's capability. It is similar to mucert, and computes the coverage *w.r.t.* a sample. It differs from mucert in that it will accept a proposed sample if the sample leads to increased coverage, and otherwise rejects the sample.

We record several metrics during the evaluation. We report the covered statements and branches by the test suites. The more program statements/branches are covered, the more validation policies can be triggered by the corresponding test suites. We also normalize the coverage Cov using the following formula

$$NormCov = \frac{Cov - \bot}{\top - \bot} \times 100\%,$$

where $\bot$ and $\top$ are respectively the lower and the upper bounds of the coverage values. We do not use the absolute coverage rates, as OpenSSL supports a rich set of functions, while many (*e.g.*, generation of self-signed certificates and private keys) do not concern certificate validation. For the same reason, we can only approximate the lower and the upper bounds, but not their actual values, which suffice for guiding certificate mutation.

We record any discovered validation discrepancies and compute the *precision* of a test suite $Cert$ as follows

$$Precision = \frac{|DCert|}{|Cert|},$$

where $|Cert|$ denotes the number of certificates in the test suite, and $DCert$ a subset of the certificates that trigger discrepancies. The more discrepancies discovered by a test suite with a fixed number of certificates, the more precise the test suite is.

We compute the *diversity* of a test suite using the formula

$$Diversity = \frac{|DDCert| + \delta}{|Cert|} \times 100\%,$$

where $|DDCert|$ denotes the number of *distinct* discrepancies, and the numerator $|DDCert| + \delta$ denotes the number of distinct encoded results ($\delta \in \{0, 1, 2\}$ to count for the all zeros and all ones if they exist). The more distinct discrepancies found, the more diverse are the certificates in the test suite.

In practice, neither discrepancies nor diversity can be conveniently computed during certificate generation, due to the very different validation styles of SSL/TLS implementations. In our evaluation, we will show that MCMC sampling does indeed help diversify a test suite, besides increasing coverage (see Section 3.3).

3.2 Results on Certificate Generation

Mucert, randmut, and greedymut all require an initial set of n certificates, but take their respective strategies to optimize the set. In the evaluation, we include in the initial set all $1,006$ certificate chains in the corpus ($n = 1,006$). When using randmut, we perform $5,000$ mutation operations, while for greedymut and mucert, we continue to mutate the certificates until the coverage does not increase for k ($k = 500/8,000$) iterations. For a straightforward, direct comparison of various methods, we produce $100,000$ frankencerts.

Table 3 and Figure 6 show the code coverage of the test suites on OpenSSL, which has 77,264 SLOC and 58,897 branches in total. We use the minimal values, 5,461 SLOC and 2,364 branches, in Table 3 to approximate the lower bound $\bot$, and the maximum values 7,714 SLOC and 3,783 branches, for the upper bound $\top$. The last column shows the normalized coverage achieved.

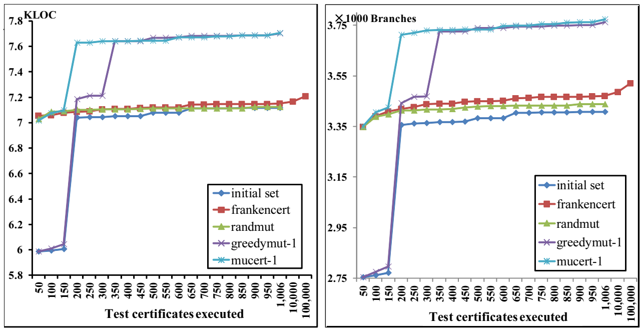
Finding 1: $1K$ *mucerts achieve up to 25% higher normalized coverage than* $1K$ *frankencerts; 200 mucerts achieve higher coverage than* $100K$ *frankencerts.*

Both mucert and greedymut achieve high coverage — 7,701-7,714 lines and 3,762-3,783 branches — regardless of the values of k and β. In contrast, frankencert and randmut achieve as low coverage as the initial set,[2] indicating that the validation code is inadequately tested by their certificates. In particular, 200 mucerts can achieve higher coverage than $100K$ frankencerts, demonstrating that mucerts are more effective in testing than frankencerts. Although mucerts and greedymut certificates achieve similar coverage values, mucerts are more diverse than the greedymut certificates, as the coverage of the former increases more rapidly than the latter.

[2] The certificates in the initial corpus cannot pass validation because they are validated against a special CA certificate, but not the certificates of their issuers.

Table 3: Coverage achieved by the test suites (*w.r.t.* OpenSSL). Greedymut-1/2 and mucert-1/2 use SOSH, while greedymut-3/4 and mucert-3/4 use BOSH. The arguments for greedymut and mucert are: (1) greedymut-1/3: $k = 500$; (2) greedymut-2/4: $k = 8,000$; (3) mucert-1: $\beta = -0.03$, $k = 500$; (4) mucert-2: $\beta = -0.03$, $k = 8,000$; (5) mucert-3: $\beta = -0.3$, $k = 500$; (6) mucert-4: $\beta = -0.3$, $k = 8,000$.

Certification generation						$\lvert$Cert$\rvert$					NormCov
approaches		Cov.	1	200	400	600	800	1006	10000	100000	($\lvert Cert\rvert = 1006 \sim 100000$)
initial test suite		stmt.	5862	7041	7050	7078	7114	7117	/		0.735
		branch	2629	3356	3368	3382	3405	3408	/		0.736
frankencert		stmt.	5518	7085	7107	7120	7146	7149	7164	7208	0.749 ~ 0.775
		branch	2409	3419	3440	3451	3467	3471	3487	3520	0.780 ~ 0.815
randmut		stmt.	5513	7099	7103	7114	7114	7122	/		0.737
		branch	2405	3414	3419	3431	3432	3438	/		0.756
Stmt. Cov. Optimized Search (SOSH)	greedymut-1	stmt.	5858	7184	7639	7670	7684	7701	/		0.994
		branch	2627	3442	3725	3740	3748	3762	/		0.985
	greedymut-2	stmt.	5858	7199	7204	7700	7713	7714 ($\top$)	/		1
		branch	2627	3454	3464	3766	3772	3774	/		0.993
	mucert-1	stmt.	5862	7627	7641	7669	7678	7704	/		0.995
		branch	2629	3713	3732	3747	3755	3774	/		0.993
	mucert-2	stmt.	6803	7663	7666	7694	7694	7702	/		0.994
		branch	3172	3736	3743	3763	3763	3770	/		0.990
Branch Cov. Optimized Search (BOSH)	greedymut-3	stmt.	5507	7607	7610	7638	7699	7707	/		0.996
		branch	2402	3704	3714	3729	3767	3776	/		0.995
	greedymut-4	stmt.	5503	7595	7645	7699	7710	7712	/		0.999
		branch	2392	3687	3737	3771	3779	3783 ($\top$)	/		1
	mucert-3	stmt.	5461 ($\perp$)	7657	7674	7703	7707	7710	/		0.998
		branch	2364 ($\perp$)	3736	3751	3770	3771	3776	/		0.995
	mucert-4	stmt.	5465	7624	7645	7696	7698	7702	/		0.994
		branch	2372	3698	3719	3760	3763	3767	/		0.988



(a) Statement coverage. **(b) Branch coverage.**

Figure 6: Coverage achieved by the test suites. The X-axis shows the numbers of executed certificates. For brevity, we omit greedymut-2/3/4 and mucert-2/3/4.

Finding 2: *Compared with the simpler SOSH, BOSH does not lead to improved coverage.*

Mucert-1/2 and Mucert-3/4 achieve similar statement and branch coverage values, which shows that even the simpler SOSH can help achieve high code coverage when the test suite has been sufficiently mutated and diversified.

Finding 3: *Greedymut/mucert generate certificates more slowly.*

Table 4 compares the time spent by different approaches on generating certificates. As the table shows, frankencert generates certificates quickly, since it adopts the simple synthesis strategy for test certificate generation. Greedymut and mucert spend much more time on generating certificates, since they need to compute the coverage of each sample for guiding the acceptance/rejection of the next sample. Greedymut-3/4 and mucert-3/4 spend on average 3.16× more time than greedymut-1/2 and mucert-1/2 at each iteration, because when BOSH is employed, LCOV needs to generate and merge large tracing files containing branch coverage information.

Table 4: Time spent on generating certificates, and iterations. The time budget is 4 days (*i.e.*, 345,600 seconds).

	#iterations	time (seconds)
frankencert	100,000	369
randmut	5,300	823
greedymut-1	1,263	12,846
greedymut-2	9,236	96,247
greedymut-3	2,297	98,780
greedymut-4	11,792	*timeout*
mucert-1	1,179	7,327
mucert-2	8,883	105,950
mucert-3	2,299	81,088
mucert-4	9,116	*timeout*

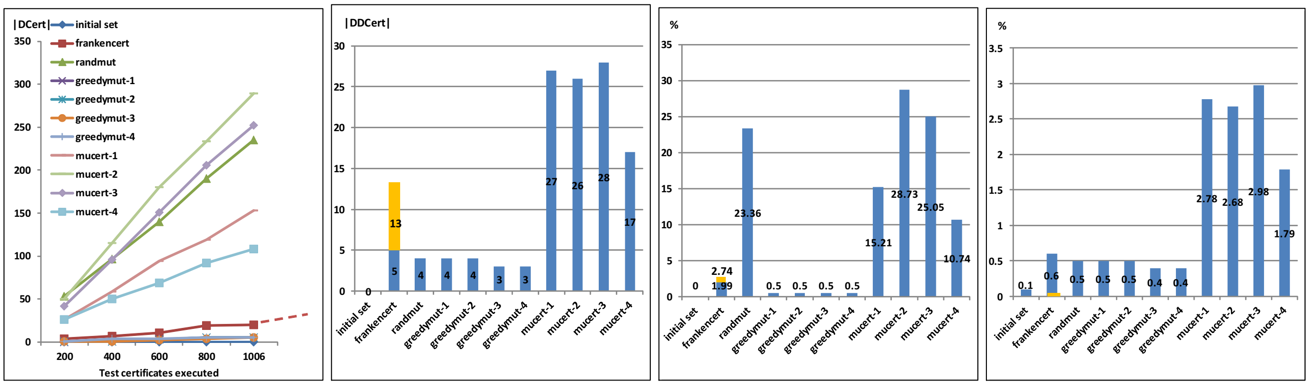
Both greedymut and mucert reach a steady state (when the coverage does not increase) after 679~3,792 (*i.e.*, $\#iterations - k$) iterations. When using SOSH, greedymut and mucert reach their steady states in up to 3.58 ($=\frac{9,236-8,000}{9,236} \times \frac{96,247}{3,600}$) hours, while when using BOSH, they reach their steady states in more than 17.6 ($=\frac{2,299-500}{2,299} \times \frac{81,088}{3,600}$) hours.

3.3 Results on Discrepancy Analysis

Table 5 shows the discrepancies found in testing. The last two columns list the precisions and diversities of different approaches.

Table 5: Discrepancies discovered by the respective test certificates.

	200	400	600	800	1,006	10K	100K	#all_accepted (111111111)	#all_rejected (000000000)	|DDCert|	Precision (%)	Diversity (%)
initial set	0	0	0	0	0	/	/	0	1,006	0	0	0.10
frankencert	4	7	11	19	20	264	2,747	0	986 ∼ 97,253	5 ∼ 13	1.9 ∼ 2.7	0.01 ∼ 0.60
randmut	53	97	140	190	235	/	/	0	766	4	23.4	0.50
greedymut-1	0	1	3	5	5	/	/	0	1,001	4	0.5	0.50
greedymut-2	1	1	3	5	5	/	/	0	1,001	4	0.5	0.50
greedymut-3	1	1	2	4	5	/	/	0	1,002	3	0.5	0.40
greedymut-4	1	4	4	5	5	/	/	0	1,002	3	0.5	0.40
mucert-1	26	59	94	119	153	/	/	0	853	27	15.2	2.78
mucert-2	50	115	180	234	289	/	/	0	717	26	28.7	2.68
mucert-3	42	96	151	206	252	/	/	1	753	28	25.0	2.98
mucert-4	26	50	69	92	108	/	/	0	898	17	10.7	1.79



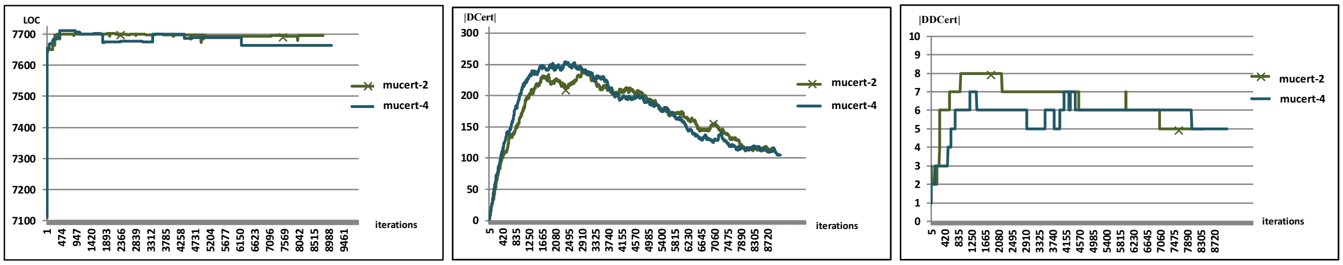
(a) Discrepancies.

(b) Distinct discrepancies.

(c) Precision.

(d) Diversity.

Figure 7: Discrepancy analysis. The columns in orange hold the values for frankencerts when $|Cert| = 100,000$.



(a) Coverage

(b) Discrepancy

(c) Distinct Discrepancies

Figure 8: Correlation between coverage and discrepancies. The X-axis stands for the numbers of iterations.

Table 6: Validation results of the 357 mucerts. A certificate is accepted by Firefox, Chrome, or IE if it can be imported into the certificate manager without any warning messages. Any parsing error reported by browsers is taken as a rejection.

	OpenSSL	PolarSSL	Gnutls	NSS	CyaSSL	MatrixSSL	Firefox	Chrome	IE
Accepted	120	344	10	9	262	299	308	2	109
Not parsable	/	12	12	38	/	23	/	/	/
Rejected	237	1	335	310	95	35	49	355	248

Figures 7(a) and (b) compare the discrepancies and distinct ones found by the different approaches, respectively; (c) and (d) compare their precisions and diversities, respectively.

Finding 4: *$1K$ mucerts trigger up to 14.5× as many discrepancies and 5.6× as many distinct discrepancies as $1K$ frankencerts; $1K$ mucerts trigger up to 2.2× as many distinct discrepancies as $100K$ frankencerts.*

Randmut and mucert trigger the most number of discrepancies when $|Cert| = 1,006$. On average, 20% of mucerts can trigger discrepancies, and each mucert-mutated test suite can help identify about 25 distinct discrepancies. In comparison, 2.8% of $100K$ frankencerts can only trigger 13 distinct discrepancies, as most frankencerts are rejected due to trivial parsing errors. In particular, $1K$ mucert-2 certificates can trigger 14.5× as many discrepancies as $1K$ frankencerts; $1K$ mucert-3 certificates can trigger 5.6× and 2.2× as many distinct discrepancies as $1K$ and $100K$ frankencerts, respectively. Many discrepancies have also been discovered by the randmut certificates, but they are reduced to only four distinct discrepancies. This shows that most discrepancies found by randmut certificates are redundant.

Finding 5: *Mucerts achieve 28.7% precision and 3.0% diversity; all distinct discrepancies found by frankencerts and randmut/greedymut certificates are also found by mucerts.*

Up to 28.7% of mucerts reveal discrepancies, so do 2.7% frankencerts, 23.4% randmut certificates and 0.5% greedymut certificates. Mxucerts achieve up to 3.0% diversity, while the others achieve only less than 0.6%. These results show that mucert is much more cost-effective than frankencert and greedymut. In addition, we have observed that each distinct discrepancy found by frankencerts and randmut/greedymut certificates is also discovered by mucerts.

Finding 6: *Mucert can stochastically diversify a test suite even when it does not further increase test coverage.*

We have investigated the samples generated by mucert-2/4, and observed how the coverage and discrepancies vary *w.r.t.* the number of iterations. To make the analysis feasible, we selected one sample every five iterations, and only ran these test suites on OpenSSL, PolarSSL, Gnutls, and NSS and identified validation discrepancies.

Figure 8(a) shows that a test suite can approach a peak coverage value after about 400 iterations. Nevertheless, MCMC sampling continues to diversify the test suite: Figures 8(b) and (c) show that more (distinct) discrepancies can be triggered by the samples even if their coverage does not increase (or even decrease).

In addition, we observe that most discrepancies (and distinct ones) are discovered by the samples between 1,200 and 2,500 iterations, but the numbers decrease slowly in subsequent iterations. One main reason for this is that the proposed distribution of MCMC sampling is in fact not symmetric: mucert needs to parse a certificate and then mutate it, so a valid certificate can be mutated to a mutant with parsing errors, but the opposite direction is usually infeasible. For example, $1K$ mucert-2 certificates can contain 159 (15.8%) certificates with parsing errors (*w.r.t.* OpenSSL) after 8,000 iterations.

3.4 Bug Reports and Developer Feedback

We have reported 20+ distinct discrepancies and an additional 357 discrepancy-triggering mucerts to SSL/TLS developers/maintainers. These certificates illustrate prevalent validation discrepancies among the different implementations. As Table 6 shows, OpenSSL does not report any parsing errors during validation, as the certificates were created by OpenSSL. PolarSSL tends to accept more certificate chains having self-signed certificates than OpenSSL, Gnutls, and NSS. 234 certificates are accepted by both MatrixSSL and CyaSSL, and 30 rejected by both, while the remaining 93 are accepted by one and rejected by the other. Browsers also behave differently when certificates are imported, 49 (14%) and 248 (69%) certificates can be imported into Firefox and IE's certificate managers without triggering any warning messages, respectively, while Chrome accepts nearly all certificates except one expired and another failed to parse.

Finding 7: *Certificate validation discrepancies are prevalent.*

However, this does not necessarily indicate that SSL/TLS connections are insecure, because the discrepancies more likely exhibit compatibility issues among SSL/TLS implementations. For example, certificates in the test suite are issued by an issuer with a valid private key, and thus the certificate chains can pass validation if they are well-formed and have not expired. But some SSL/TLS implementations also validate malformed certificate chains. To date, we have yet to find any real exploits that maliciously use the discrepancies we have found, although it has been reported that parsing differences between CA software and browser certificate validation code can be exploited as man-in-the-middle attacks [32].

We have also made some fake certificates issued by a valid CA certificate with a fake private key, and checked whether they could trigger any discrepancies. They are similar to some real man-in-the-middle attacks.

Finding 8: *Developers do have many doubts and concerns when implementing certificate validation code.*

OpenSSL security team admitted that there are a lot of certificates that can violate RFCs, but they cannot reject all of them unless they are certainly security related.

PolarSSL 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 in order to cover that case. They have fixed the naming problem in their internal development branch and the patch will be included in the next release.

OpenSSL, GNUTLS, PolarSSL, and MatrixSSL behave differently when a certificate has two instances of the `SubjectKeyIdentifier`. This finding has triggered an open question on IETF PKIX,[3] 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[4] 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.

PolarSSL 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/CyaSSL) 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). PolarSSL developers have explained that PolarSSL accepts these certificate chains since they have complete and valid key chains; they may reject these chains in future development.

[3] "Suggested replacement text for RFC5280, section 4.2.1.1," `http://www.ietf.org/mail-archive/web/pkix/current/msg33248.html`

[4] `https://tls.mbed.org/tech-updates/releases/mbedtls-1.3.10-released`

OpenSSL, PolarSSL, 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*" [21]. 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 not to do it. More importantly, accepting such chains allows users to trust self-signed non-CA certificates if they choose to.

All the SSL/TLS implementations and browsers but NSS strictly reject the invalid certificates issued by a valid CA with a fake private key. We have reported to the NSS developers that `certutil` incidentally skips signature checking when it validates a certificate in a certificate database. The developers have responded to us that in PKI, existence as trusted is sufficient for validity. On the other hand, other developers have claimed that this behavior confuses many users and even security engineers.

The Chrome security team has explained that importing an invalid certificate into the certificate manager is intentional and confers trust. When an end user attempts to manually import an untrusted certificate into the certificate manager, certificate validation is skipped.

4. RELATED WORK

We discuss four strands of related work: (1) testing certificate validation code, (2) random and symbolic execution, (3) feedback-directed test generation, and (4) MCMC sampling for testing.

Testing Certificate Validation Code SSL/TLS tools and libraries are vulnerable. Marlinspike [33, 34] has shown several vulnerabilities in certificate validation code in browsers and SSL libraries. Kaminsky *et al.* [32] have shown that a CA can issue a certificate that can be used for man-in-the-middle attacks due to parsing differences between CA software and browser certificate validation code. Georgiev *et al.* have used both white- and black-box techniques to uncover vulnerabilities in validation logic [28], and have identified several vulnerabilities. In contrast, mucert does not require deep security knowledge, but rather employs a set of test certificates to test the certificate validation code and find flaws inside. Thus the mucert approach can be easily adapted in practice for verifying SSL/TLS tools and libraries.

Bates *et al.* [12] present CertShim, a mechanism that improves SSL security by interposing on SSL APIs and retrofitting legacy software to support SSL trust enhancements. They also show how to poll results of several verification methods to improve security. Mucert also leverages results of multiple certificate verification handlers, but focus on generating test certificates to trigger discrepancies among these handlers.

The most closely related work to ours is frankencert [13], the first systematic technique for synthesizing test certificates for testing SSL/TLS implementations. Due to its completely random and thus "blind" nature, most of frankencerts are invalid or redundant, and do not trigger new validation policies. Compared with frankencert, mucert allows the tester to much more effectively explore the input space and select diverse certificates for testing.

Random and Symbolic Execution Random testing has been a very common testing technique, which requires testers to select tests from an input domain at random. Random testing is usually not cost effective, especially for settings with structurally complex input domains. Adaptive Random Testing (ART) is a controversial idea that tries to spread out the selected values over the input domain [15–17, 20]. Many ART algorithms mainly select tests based on the locations of successful tests, and use distances to measure whether the next test case is sufficiently far away from all successful tests. Meanwhile, ART is not applicable in our adversarial testing, as the input space of X.509 certificate is high dimensional, while it has been reported that ART often does not work well even in one-dimension domains [11].

Symbolic execution has been used for generating test cases for complex programs. KLEE [14] automatically generates test cases by running programs symbolically and generating path constraints, attempting to hit every line of executable code and detect dangerous operations. Dynamic symbolic execution techniques, such as DART [29], CUTE [43], and Pex [44], improve the effectiveness of symbolic execution and random testing by dynamically analyzing the program behaviors under random testing and automatically generating new test inputs to direct the execution along alternative program paths. However, so far these symbolic execution tools have not been used for generating certificates, as the modern constraint solvers are not able to solve the intricate syntactical and semantic constraints on the certificates. In contrast, mucert proposes coverage optimized search heuristics to mutate certificates, but does not rely on any constraint solver to generate certificates.

Feedback-directed Test Generation Another emerging direction is to employ the previously constructed tests to design new tests. Pacheco *et al.* present a feedback-directed random test generation technique [37–39]. The feedback-directed technique builds inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously constructed inputs. Once an input is built, it is executed, and the result determines whether the input should be rejected or accepted for generating more inputs. Inspired by the idea of feedback-directed testing, we leverage the code coverage to measure and optimize a test suite.

Fraser and Arcuri introduce a technique called *whole test suite generation* that can evolve all the test cases in a test suite simultaneously [26]. The technique starts with an initial set of randomly generated test suites, and then uses a genetic algorithm to optimize toward satisfying a chosen coverage criterion, while keeping the total size of the test suite as small as possible. Mucert also employs coverage to measure a test suite, but uses MCMC sampling to stochastically diversify a test suite, as the diversities of test suites cannot be rapidly obtained during the sampling phase.

MCMC Sampling for Testing Zhou *et al.* propose a Markov chain Monte Carlo Random Testing (MCMCRT) approach to random testing [45]. On the basis of the Bayes approach to parametric models for software testing, MCMCRT can utilize the prior knowledge and the information on preceding test outcomes for their parameter estimation. The MCMC sampler is also used to enhance performance of random testing [47] and test case prioritization [46]. However, MCMCRT mainly generates test inputs for numerical programs. To our knowledge, our work is the first that effectively uses MCMC sampling for creating structured test inputs.

5. CONCLUSION

We have introduced mucert, a guided differential testing of certificate validation, by adopting MCMC sampling to mutate X.509 certificates and produce diverse certificates for testing SSL/TLS implementations. Our experimental results have clearly demonstrated mucert's strengths over frankencert — more detected discrepancies and improved code coverage with orders of magnitude fewer test certificates. We believe that developers can use mucert routinely to identify latent flaws in SSL/TLS implementations to improve the security and robustness of the Internet infrastructure.

6. REPLICATION PACKAGE

The mucert package has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations. The mucert package is composed of three key components:

1. *Certificate mutation engine* that consumes a set of Internet certificates and produces another set of diversified test certificates. Five certificate optimization algorithms are supported in the mutation engine: the MCMC-guided certificate mutation algorithm proposed in this paper, and two random and two greedy mutation algorithms that we designed to compare against mucert;
2. *Scripts* for supporting differential testing and analysis; and
3. *Seeding certificates* and a *sample CA certificate*; for comparison reasons, we used the same seeding certificates provided by frankencert [13].

A typical certificate generation process includes two phases:

1. *Preparation*: Mucert generates a coverage trace file for each certificate (*w.r.t.* OpenSSL) and constructs a coverage information tree for the corpus of seeding certificates; and
2. *Mutation*: Mucert takes one optimization algorithm to mutate the seeding certificates. The mutants can be either accepted for further mutations or discarded, and the coverage information tree is updated along with the mutations.

After a number of iterations, the resulting certificates are used to differentially test the SSL/TLS implementations. Most of the state-of-the-art SSL/TLS implementations and web browsers can be tested in C-S mode (*e.g.*, OpenSSL, PolarSSL/mbed TLS, GnuTLS, CyaSSL, NSS, MatrixSSL, Google's Chrome, and Mozilla's Firefox). Some SSL/TLS implementations (*e.g.*, OpenSSL, PolarSSL, and GnuTLS) can also validate these certificate files via the command line. Some basic commands for validating certificates are shown next, where `$ca_file` is a CA certificate and `$cert` a mutated certificate to be validated:

- OpenSSL: `openssl verify -CAfile $ca_file $cert`
- PolarSSL/mbed TLS: `cert_app mode='file' filename=$cert ca_file=$ca_file`
- GnuTLS: `certtool --verify --load-ca-certificate=$ca_file < $cert`

The validation results are recorded and compared within a spreadsheet. Each cell records the validation result (accept or reject) of a certificate *w.r.t.* an SSL/TLS implementation. When a certificate is accepted by one SSL/TLS implementation but rejected by another, a discrepancy is found and reported.

A corpus of 357 discrepancy-triggering certificates are also contained in the replication package such that other researchers and developers can use them to further investigate the causes of discrepancies among SSL/TLS implementations.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers and the Replication Packages Evaluation Committee members for their constructive feedback. We also thank Baishakhi Ray for her invaluable guidance and suggestions on certificate validation. This research was sponsored in part by 973 Program in China (Grant No. 2015CB352203), the National Nature Science Foundation of China (Grant No. 91118004, 61272102, 61472242, and 61100051), and United States NSF Grants (Grant No. 1117603, 1319187, and 1349528). Yuting Chen is also partially supported by the China Scholarship Council (CSC).

8. REFERENCES

[1] https://www.openssl.org/.

[2] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

[3] https://polarssl.org/.

[4] http://www.gnutls.org/.

[5] http://www.yassl.com/yaSSL/Products-cyassl.html.

[6] http://www.matrixssl.org/.

[7] http://www.google.com/chrome/.

[8] https://www.mozilla.org/en-US/firefox.

[9] http://windows.microsoft.com/en-us/internet-explorer.

[10] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 181–192, 2014.

[11] A. Arcuri and L. C. Briand. Adaptive random testing: an illusion of effectiveness? In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 265–275, 2011.

[12] A. M. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, K. R. B. Butler, and A. Alkhelaifi. Securing SSL certificate verification through dynamic linking. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 394–405, 2014.

[13] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy*, pages 114–129, 2014.

[14] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[15] K. P. Chan, T. Y. Chen, and D. Towey. Restricted random testing: Adaptive random testing by exclusion. *International Journal of Software Engineering and Knowledge Engineering*, 16(4):553–584, 2006.

[16] T. Y. Chen. Adaptive random testing. In *International Conference on Quality Software (QSIC)*, page 443, 2008.

[17] T. Y. Chen, D. Huang, F. Kuo, R. G. Merkel, and J. Mayer. Enhanced lattice-based adaptive random testing. In *ACM Symposium on Applied Computing*, pages 422–429, 2009.

[18] S. Chib and E. Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49(4):327–335, Nov. 1995.

[19] S. Chokhani and W. Ford. Internet X.509 public key infrastructure certificate policy and certification practices framework. http://www.ietf.org/rfc/rfc2527.txt.

[20] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In *International Conference on Software Engineering (ICSE)*, pages 71–80, 2008.

[21] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[22] T. Dierks and C. Allen. The TLS Protocol Version 1.0, January 1999. Available from `http://www.ietf.org/rfc/rfc2246`.

[23] T. Dierks and E. Rescorla. The TLS Protocol Version 1.1, April 2006. Available from `http://www.ietf.org/rfc/rfc4346`.

[24] T. Dierks and E. Rescorla. The TLS Protocol Version 1.2, August 2008. Available from `http://www.ietf.org/rfc/rfc5246`.

[25] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *USENIX Security Symposium*, pages 605–620, 2013.

[26] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Trans. Software Eng.*, 39(2):276–291, 2013.

[27] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0, August 2011. Available from `http://www.ietf.org/rfc/rfc6101`.

[28] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security (CCS)*, pages 38–49, 2012.

[29] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[30] A. Groce, G. J. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering (ICSE)*, pages 621–631, 2007.

[31] N. Gruschka, L. L. Iacono, and C. Sorge. Analysis of the current state in website certificate validation. *Security and Communication Networks*, 7(5):865–877, 2014.

[32] D. Kaminsky, M. L. Patterson, and L. Sassaman. PKI layer cake: New collision attacks against the global X.509 infrastructure. In *International Conference on Financial Cryptography and Data Security (FC)*, pages 289–303, 2010.

[33] M. Marlinspike. IE SSL vulnerability, 2002. Available from `http://www.thoughtcrime.org/ie-ssl-chain.txt`.

[34] M. Marlinspike. Null prefix attacks against SSL/TLS certificates, 2009. Available from `http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf`.

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

[36] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[37] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *Companion to the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 815–816, 2007.

[38] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 87–96, 2008.

[39] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[40] E. Rescola. HTTP over TLS, May 2000. Available from `http://www.ietf.org/rfc/rfc2818`.

[41] P. Saint-Andre and J. Hodges. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS), March 2011. Available from `https://tools.ietf.org/html/rfc6125`.

[42] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–316, 2013.

[43] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification (CAV)*, pages 419–423, 2006.

[44] N. Tillmann and J. de Halleux. Pex: White box test generation for .NET. In *International Conference on Tests and Proofs (TAP)*, pages 134–153, 2008.

[45] B. Zhou, H. Okamura, and T. Dohi. Markov Chain Monte Carlo random testing. In *Advances in Computer Science and Information Technology*, pages 447–456, 2010.

[46] B. Zhou, H. Okamura, and T. Dohi. Application of Markov Chain Monte Carlo random testing to test case prioritization in regression testing. *IEICE Transactions*, 95-D(9):2219–2226, 2012.

[47] B. Zhou, H. Okamura, and T. Dohi. Enhancing performance of random testing through Markov Chain Monte Carlo methods. *IEEE Trans. Computers*, 62(1):186–192, 2013.