Published in final edited form as:
IEEE Secur Priv. 2014 ; 2014: 114–129.

Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations

Chad Brubaker[*,†], **Suman Jana**[†], **Baishakhi Ray**[‡], **Sarfraz Khurshid**[†], and **Vitaly Shmatikov**[†]

[*]Google

[†]The University of Texas at Austin

[‡]University of California, Davis

Abstract

Modern network security rests on the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. Distributed systems, mobile and desktop applications, embedded devices, and all of secure Web rely on SSL/TLS for protection against network attacks. This protection critically depends on whether SSL/TLS clients correctly validate X.509 certificates presented by servers during the SSL/TLS handshake protocol.

We design, implement, and apply the first methodology for large-scale testing of certificate validation logic in SSL/TLS implementations. Our first ingredient is “frankencerts,” synthetic certificates that are randomly mutated from parts of real certificates and thus include unusual combinations of extensions and constraints. Our second ingredient is differential testing: if one SSL/TLS implementation accepts a certificate while another rejects the same certificate, we use the discrepancy as an oracle for finding flaws in individual implementations.

Differential testing with frankencerts uncovered 208 discrepancies between popular SSL/TLS implementations such as OpenSSL, NSS, CyaSSL, GnuTLS, PolarSSL, MatrixSSL, etc. Many of them are caused by serious security vulnerabilities. For example, any server with a valid X.509 version 1 certificate can act as a rogue certificate authority and issue fake certificates for any domain, enabling man-in-the-middle attacks against MatrixSSL and GnuTLS. Several implementations also accept certificate authorities created by unauthorized issuers, as well as certificates not intended for server authentication.

We also found serious vulnerabilities in how users are warned about certificate validation errors. When presented with an expired, self-signed certificate, NSS, Safari, and Chrome (on Linux) report that the certificate has expired—a low-risk, often ignored error—but not that the connection is insecure against a man-in-the-middle attack.

These results demonstrate that automated adversarial testing with frankencerts is a powerful methodology for discovering security flaws in SSL/TLS implementations.

I. Introduction

Secure Sockets Layer (SSL) and its descendant Transport Layer Security (TLS) protocols are the cornerstone of Internet security. They are the basis of HTTPS and are pervasively used by Web, mobile, enterprise, and embedded software to provide end-to-end confidentiality, integrity, and authentication for communication over insecure networks.

SSL/TLS is a big, complex protocol, described semi-formally in dozens of RFCs. Implementing it correctly is a daunting task for an application programmer. Fortunately, many open-source implementations of SSL/TLS are available for developers who need to incorporate SSL/TLS into their software: OpenSSL, NSS, GnuTLS, CyaSSL, PolarSSL, MatrixSSL, cryptlib, and several others. Several Web browsers include their own, proprietary implementations.

In this paper, we focus on *server authentication*, which is the only protection against man-in-the-middle and other server impersonation attacks, and thus essential for HTTPS and virtually any other application of SSL/TLS. Server authentication in SSL/TLS depends entirely on a single step in the handshake protocol. As part of its “Server Hello” message, the server presents an X.509 certificate with its public key. The client must validate this certificate. **Certificate validation** involves verifying the chain of trust consisting of one or more certificate authorities, checking whether the certificate is valid for establishing SSL/TLS keys, certificate validity dates, various extensions, and many other checks.

Systematically testing correctness of the certificate validation logic in SSL/TLS implementations is a formidable challenge. We explain the two main hurdles below.

First problem: generating test inputs

The test inputs, i.e., X.509 certificates, are structurally complex data with intricate semantic and syntactic constraints. The underlying input space is huge with only a tiny fraction of the space consisting of actual certificates. A simple automated technique, such as random fuzzing, is unlikely to produce more than a handful of useful inputs since a random string is overwhelmingly unlikely to even be parsable as a certificate.

Some test certificates can be created manually, but writing just a small suite of such complex inputs requires considerable effort; manually creating a high-quality suite is simply infeasible. Furthermore, the testing must include “corner cases”: certificates with unusual combinations of features and extensions that do not occur in any currently existing certificate but may be crafted by an attacker.

Second problem: interpreting the results of testing

Given a test certificate and an SSL/TLS implementation, we can record whether the certificate has been accepted or rejected, but that does not answer the main question: is the implementation correct, i.e., is the accepted certificate valid? And, if the certificate is rejected, is the reason given for rejection correct?

Manually characterizing test certificates as valid or invalid and writing the corresponding assertions for analyzing the outputs observed during testing does not scale. A naive approach to automate this characterization essentially requires re-implementing certificate validation, which is impractical and has high potential for bugs of its own. Interpreting the results of large-scale testing requires an *oracle* for certificate validity.

Our contributions

We design, implement, and evaluate the first approach for systematically testing certificate validation logic in SSL/TLS implementations. It solves both challenges: (1) automatically generating test certificates, and (2) automatically detecting when some of the implementations do not validate these certificates correctly.

The first step of our approach is **adversarial input generation**. By design, our generator synthesizes test certificates that are syntactically well-formed but may violate many of the complex constraints and internal dependencies that a valid certificate must satisfy. This enables us to test whether SSL/TLS implementations check these constraints and dependencies.

To “seed” the generator, we built a corpus of 243,246 real SSL/TLS certificates by scanning the Internet. Our generator broke them down into parts, then generated over 8 million **frankencerts** by mutating random combinations of these parts and artificial parts synthesized using the ASN.1 grammar for X.509. By construction, frankencerts are parsable as certificates, yet may violate X.509 semantics. They include unusual combinations of critical and non-critical extensions, rare extension values, strange key usage constraints, odd certificate authorities, etc. Testing SSL/TLS implementations with frankencerts exercises code paths that rarely get executed when validating normal certificates and helps elicit behaviors that do not manifest during conventional testing.

Our second insight is that multiple, independent implementations of X.509 certificate validation—the very same implementations that we are testing—can be used as an **oracle to detect flaws in validation logic**. For each frankencert, we compare the answers produced by OpenSSL, NSS, GnuTLS, CyaSSL, PolarSSL, MatrixSSL, OpenJDK, and Bouncy Castle. These SSL/TLS libraries are supposed to implement the same certificate validation algorithm and, therefore, should agree on every certificate. Differences in the implementations of functionality left unspecified by the X.509 standard may cause a “benign” discrepancy, but most discrepancies mean that some of the disagreeing SSL/TLS implementations are incorrect.

Our differential mutation testing of SSL/TLS implementations on 8,127,600 frankencerts uncovered 208 discrepancies between the implementations, many of which are caused by serious flaws. For example, MatrixSSL silently accepts X.509 version 1 certificates, making all MatrixSSL-based applications vulnerable to man-in-the-middle attacks: anyone with a valid version 1 certificate can pretend to be an intermediate certificate authority (CA), issue a fake certificate for any Internet domain, and that certificate will be accepted by MatrixSSL.

In GnuTLS, our testing discovered a subtle bug in the handling of X.509 version 1 certificates. Due to a mismatch between two flags, the code that intends to accept only locally trusted version 1 root certificates is actually accepting *any* version 1 CA certificate, including fake ones from malicious servers. This bug could not have been found without frankencerts because it is not triggered by any real certificate from our corpus (but, of course, a man-in-the-middle attacker could craft a malicious certificate to exploit this vulnerability).

Many vulnerabilities are caused by incorrect or missing checks on the restrictions that root CAs impose on lower-level CAs. MatrixSSL does not check path length constraints. If a restricted CA (e.g., a corporate CA whose authority only extends to a particular enterprise) creates a new intermediate CA, who then issues certificates for any Internet domain, these certificates will be accepted by MatrixSSL. GnuTLS, CyaSSL, and PolarSSL do not check key usage constraints. As a consequence, an attacker who compromises the code signing key of some company can use it to spoof that company's servers in TLS connections. Most of these flaws could not have been discovered without frankencerts because incorrect validation logic is only triggered by certificates of a certain form, not by "normal" certificates.

Even if an SSL/TLS implementation correctly rejects a certificate, the reason given to the user is very important because Web browsers and other interactive applications often allow the user to override the warning. For example, if the warning is that the certificate expired yesterday, this may indicate a lazy system administrator but does not imply that the connection is insecure. Because the risk is low, the user may click through the warning. If, on the other hand, the certificate is not issued by a legitimate certificate authority, this means that the server could have been impersonated and the connection may be insecure.

Our differential testing uncovered serious vulnerabilities in how SSL/TLS implementations report errors. When presented with an expired, self-signed certificate, NSS reports that the certificate has expired but not that the issuer is invalid. This vulnerability found its way into Web browsers such as Chrome on Linux and Safari. Since users tend to click through expired-certificate warnings—and are advised to do so [1]—this flaw gives attackers an easily exploitable vector for man-in-the-middle attacks against all users of these Web browsers.

In summary, adversarial test input generation and differential mutation testing on millions of "frankencerts" synthesized from parts of real certificates is a powerful new technique for uncovering deep semantic errors in the implementations of SSL/TLS, the most important network security protocol.

II. Related work

A. Security of SSL/TLS implementations

We are not aware of any prior work on systematic, automated discovery of certificate validation vulnerabilities in the implementations of SSL/TLS clients.

Moxie Marlinspike demonstrated several flaws in the implementations of SSL/TLS certificate validation [55, 56, 57], including the lack of CA bit checking in Microsoft's CryptoAPI as of 2002 [54]. More recently, the same vulnerability was discovered in the SSL implementation on Apple iOS [40].

Georgiev et al. carried out a study of certificate validation vulnerabilities caused by the incorrect *use* of SSL/TLS APIs, as opposed to flaws in the implementations of these APIs [31]. Georgiev et al. focus primarily on the incorrect validation of hostnames in server certificates at a different level in the software stack—in applications, transport libraries, and Web-services middleware. Fahl et al. analyzed incorrect usage of SSL in Android apps [29]. The class of certificate validation vulnerabilities analyzed in this paper is complementary to and has little overlap with the vulnerabilities discovered in [29, 31]. Unlike [29, 31], we developed an automated technique for discovering certificate validation vulnerabilities.

A survey of security issues in SSL/TLS can be found in [16]. Cryptographic flaws in SSL/TLS implementations and the protocol itself—including compression, initialization, padding of cipher modes and message authentication codes, etc.—can be exploited to attack *confidentiality*, especially when the protocol is used for HTTPS (HTTP over SSL) [3, 24, 72]. By contrast, this paper is about *authentication* flaws.

Flaws in SSL server implementations can be exploited for chosen-ciphertext attacks, resulting in private key compromise [8, 9]. Flaws in pseudo-random number generation can produce SSL/TLS keys that are easy to compromise [38, 50].

Hash collisions [77] and certificate parsing discrepancies between certificate authorities (CAs) and Web browsers [44] can trick a CA into issuing a valid leaf certificate with the wrong subject name, or even a rogue intermediate CA certificate. By contrast, we focus on verifying whether SSL/TLS implementations correctly handle *invalid* certificates.

Large-scale surveys of SSL certificates "in the wild" can be found in [19, 25, 27, 78]. Because their objective is to collect and analyze certificates, not to find certificate validation errors in SSL/TLS implementations, they are complementary to this paper: for example, their certificate corpi can be used to "seed" frankencert generation (Section VII). Delignat-Lavaud et al. note that GnuTLS ignores unsupported critical extensions [19], matching what we found with automated testing.

Akhawe et al. surveyed SSL warnings in Web browsers [1]. One of their recommendations is to accept recently expired certificates. As we show in Section IX, several Web browsers show just the "Expired certificate" warning even if the expired certificate is not issued by a trusted CA and the connection is thus insecure. Akhawe and Felt performed a large-scale user study of the effectiveness of browser security warnings [2]. One of their findings is that users are *less* likely to click through an "Expired certificate" warning than through an "Untrusted issuer" warning, possibly because the former tend to occur at websites that previously did not produce any warnings. Amann et al. demonstrated that certain signs of man-in-the-middle attacks, such as certificates never seen before for a given domain or issued by an unusual CA, can be caused by benign changes in the CA infrastructure [4]. SSL security indicators in mobile Web browsers were studied in [5, 6].

The focus of this paper is on server certificate authentication, which is the most common usage pattern for SSL certificates. The other direction, i.e., client certificate authentication, was analyzed in [21, 60]. Our adversarial testing techniques for finding bugs in the client-side validation of server certificates can also be applied to the implementations of server-side validation of client certificates.

Several recent high-profile vulnerabilities highlighted the need for thorough security analysis of SSL/TLS implementations. The implementation of the SSL/TLS handshake in Mac OS and iOS accidentally did not check whether the key used to sign the server's key exchange messages matches the public key in the certificate presented by the server, leaving this implementation vulnerable to server impersonation [49] (this vulnerability is not caused by incorrect certificate validation). In GnuTLS, certain errors during certificate parsing were accidentally interpreted as successful validation, thus enabling server impersonation [33]. We discuss the latter vulnerability in more detail in Section VIII.

B. Software testing

Our work introduces a novel *black-box* testing approach to address two foundational software testing problems—*generation* of test inputs and *validation* of program outputs (aka the "oracle" problem)—in the context of finding security bugs, specifically in SSL/TLS implementations. Researchers have extensively studied these two problems over the last few decades in a number of contexts and developed various automated techniques to address them. For example, techniques using grammars [48, 52, 58, 75, 79], constraints [13, 53], dedicated generators [18], fuzzing [36], symbolic execution [12, 35, 45, 47, 74], and genetic algorithms [7] provide automated generation of inputs for black-box and white-box testing, while techniques using correctness specifications [15], differential testing [59], and metamorphic testing [14] provide automated validation of program outputs. Differential black-box testing has been successfully used to find parsing discrepancies between antivirus tools that can help malware evade detection [42].

The use of **grammars** in testing dates back to the 1970s [62] and has provided the basis for randomized [52, 58, 75, 79] and systematic [48] techniques for finding application bugs. The most closely related work to ours is Yang et al.'s Csmith framework, which used random grammar-based generation of C programs to discover many bugs in production C compilers [79]. The key difference between Csmith and our work is input generation. Csmith uses purely grammar-based generation without actual C programs and hence only produces input programs with language features that are explicitly supported by its generation algorithm. Moreover, the design goal of Csmith is to generate *safe* programs that have a unique meaning and no undefined behaviors. This allows Csmith to use a straightforward test oracle that performs identity comparison on outputs for differential testing. By contrast, our goal is to explore behaviors of SSL/TLS implementations that are not exercised by valid certificates and thus more likely to contain security bugs. Hence, our test generator does not need to ensure that test outputs conform to a restricted form. To detect validation errors, we *cluster* certificates into "buckets" based on the outputs produced by each SSL/TLS implementation when presented with a given certificate, with each bucket representing a discrepancy between the implementations. As explained in Section IX, multiple discrepancies may be

caused by the same underlying implementation error (in our testing, 15 root causes led to 208 discrepancies).

Clustering test executions is a well-explored area, e.g., to diagnose the causes of failed executions by reducing the number of failures to inspect [32, 41, 43, 61] or to distinguish failing and passing executions in the context of a *single* implementation [20]. We use clustering and differential testing in tandem to identify incorrect behavior in the context of *multiple* implementations tested together.

Our test input generator combines parts of existing real certificates and also injects synthetic artificial parts using operations that resemble *combination* and *mutation* in **genetic algorithms** [39]. In principle, it may be possible to define a genetic algorithm for certificate generation by customizing genetic combination and mutation with respect to the SSL certificate grammar, fields, their values, extensions, etc. The main challenge for effective genetic search is how to define an appropriate *fitness function*, which must measure the potential usefulness of a candidate input. Genetic search, as well as other heuristics for test input generation, can complement systematic exploration using guided sampling [7].

The classic idea of **symbolic execution** [47] as well as its more recent variants, e.g., where concrete inputs guide symbolic execution [12, 35, 74], enable a form of white-box test input generation that has received much recent attention for finding security bugs [36, 37, 46, 73]. Godefroid et al.'s SAGE [36] introduced *white-box fuzzing* that executes a given suite of inputs, monitors their execution paths, and builds symbolic path condition constraints, which are systematically negated to explore their neighboring paths. SAGE found several new bugs in Windows applications, including media players and image processors. Grammar-based whitebox fuzzing [34] uses a grammar to enumerate valid string inputs by solving constraints over symbolic grammar tokens. A security-focused application using a context-free fragment of the JavaScript grammar to test the code generation module of the Internet Explorer 7 JavaScript interpreter showed that the use of the grammar provides enhanced code coverage. Similar but independent work on CESE [51] uses *symbolic grammars* with symbolic execution to create higher-coverage suites for select UNIX tools, albeit in a non-security setting.

Kiezun et al.'s Ardilla [46] uses concolic execution to generate test inputs that drive its dynamic taint analysis and mutates the inputs using a library of attack patterns to create SQL injection and cross-site scripting attacks. Halfond et al. [37] show how symbolic execution can more precisely identify parameter values that define the interfaces of Web applications, and facilitate finding vulnerabilities. Saxena et al.'s Kudzu [73] uses a symbolic execution framework based on a customized string constraint language and solver to find code injection vulnerabilities in JavaScript clients.

Brumley et al. [10] proposed a white-box symbolic analysis technique to guide differential testing [59]. Their analysis is driven by concrete executions in the spirit of dynamic symbolic (aka concolic) execution [12, 35, 74]. They use *weakest preconditions* [23] over select execution paths together with constraint solving to compute inputs that likely cause

parsing discrepancies between different implementations of protocols such as HTTP and NTP.

There are two basic differences between our methodology and that of [10]. First, our black-box approach does not require analyzing either the source, or the binary code. Second, the need to solve path constraints limits the scalability of the approach described in [10]. Generating even a single test certificate using their technique requires symbolic analysis of both the parsing code and the certificate validation code hidden deep inside the program. SSL certificates are structurally more complex than HTTP and NTP inputs, and, crucially, the certificate validation logic lies deeper in SSL/TLS implementations than the X.509 parsing code. For example, a MiniWeb server responding to a GET /index.html request (one of the case studies in [10]) executes 246,910 instructions. By contrast, the simplest of our test cases—an OpenSSL client processing a certificate chain of length 1 with zero extensions—executes 27,901,961 instructions.

An interesting avenue for future research is to explore whether the two approaches could be used in conjunction and, in particular, whether generation of test SSL certificates can benefit from the fact that the technique of [10] performs a *directed* search for likely behavioral differences.

More recent work by Ramos and Engler on UC-KLEE [63], which integrates KLEE [11] and lazy initialization [45], applies more comprehensive symbolic execution over a bounded exhaustive execution space to check code equivalence; UC-KLEE has been effective in finding bugs in different tools, including itself. In principle, such *goal-directed* approaches are very powerful: they integrate the spirit of differential testing with symbolic analysis to create formulas that explicitly capture behavioral differences of interest. However, the resulting formulas in the context of structurally complex data can be exceedingly complex since they represent destructive updates in imperative code using a stateless logic. Scaling such approaches to SSL/TLS implementations is an open problem.

In summary, while approaches based on symbolic execution have been successful in finding bugs in many applications, their central requirement—the need to solve constraints for each execution path explored in symbolic execution—is the basic bottleneck that limits their scalability and applicability for programs that operate on complex data types, such as the structurally complex SSL certificates, and have complex path conditions that can be impractical to solve. By contrast, **our test generation algorithm is not sensitive to the implementation-level complexity of the programs being tested.** Instead, it focuses on the systematic exploration of the space of likely useful inputs and thus reduces the overall problem complexity by de-coupling the complexity of the input space from that of the SSL/TLS implementations.

Srivastava et al. [76] use *static* differential analysis, which does not perform test generation or execution, to analyze consistency between different implementations of the Java Class Library API and use the discrepancies as an oracle to find flaws in the implementations of access-control logic. While static analysis and dynamic analysis, such as testing, are well-known to have complementary strengths, they can also be applied in synergy [28]. For

example, for testing SSL/TLS implementations, static dataflow analysis could potentially reduce the space of candidate inputs for the test generator by focusing it to exercise fewer values or fewer combinations of values for certain certificate extensions.

III. Overview of SSL/TLS

A. SSL/TLS protocol

The Secure Sockets Layers (SSL) Protocol Version 3.0 [70] and its descendants, Transport Layer Security (TLS) Protocol Version 1.0 [64], Version 1.1 [67], and Version 1.2 [68], are the "de facto" standard for secure Internet communications. The primary goal of the SSL/TLS protocol is to provide privacy and data integrity between two communicating applications.

In this paper, we focus on a particular security guarantee promised by SSL/TLS: *server authentication*. Server authentication is essential for security against network attackers. For example, when SSL/TLS is used to protect HTTP communications (HTTPS), server authentication ensures that the client (e.g., Web browser) is not mistaken about the identity of the Web server it is connecting to. Without server authentication, SSL/TLS connections are insecure against *man-in-the-middle attacks*, which can be launched by malicious Wi-Fi access points, compromised routers, etc.

The SSL/TLS protocol comprises the *handshake protocol* and the *record protocol*. Server authentication is performed entirely in the handshake protocol. As part of the handshake, the server presents an X.509 certificate with its public key [69]. The client must *validate* this certificate as described in Section IV. If the certificate is not validated correctly, authentication guarantees of SSL/TLS do not hold.

Certificate validation in SSL/TLS critically depends on *certificate authorities* (CAs). Consequently, we analyze the correctness of SSL/TLS implementations under the assumption that the CAs trusted by the client correctly verify the identities of the servers to whom they issue certificates. If this assumption does not hold—e.g., a trusted CA has been compromised or tricked into issuing false certificates [17, 22]—SSL/TLS is not secure regardless of whether the client is correct or not.

In summary, we aim to test if the implementations of SSL/TLS clients correctly authenticate SSL/TLS servers in the presence of a standard "network attacker," who can control any part of the network and run his own servers, possibly with their own certificates, but does not control legitimate servers and cannot forge their certificates.

B. SSL/TLS implementations

In this paper, we focus primarily on testing open-source implementations of SSL/TLS. Our testing methodology can be successfully applied to closed-source implementations, too (as illustrated by our testing of Web browsers), but having access to the source code makes it easier to identify the root causes of the flaws and vulnerabilities uncovered by our testing.

We tested the following SSL/TLS implementations: OpenSSL, NSS, CyaSSL, GnuTLS, PolarSSL, MatrixSSL, cryptlib, OpenJDK, and Bouncy Castle. These implementations are distributed as open-source software libraries so that they can be incorporated into applications that need SSL/TLS for secure network communications.

Many vulnerabilities stem from the fact that applications use these libraries incorrectly [31], especially when some critical part of SSL/TLS functionality such as verifying the server's hostname is delegated by the SSL/TLS library to the application. In this paper, however, we focus on flaws *within* the libraries, not in the applications that use them, with one exception —Web browsers.

HTTPS, the protocol for protecting Web sessions from network attackers, is perhaps the most important application of SSL/TLS. Therefore, we extend our testing to Web browsers, all of which must support HTTPS: Firefox, Chrome, Internet Explorer, Safari, Opera, and WebKit (the latter is a browser "engine" rather than a standalone browser). Web browsers typically contain proprietary implementations of SSL/TLS, some of which are derived from the libraries listed above. For example, Firefox and Chrome use a version of NSS, while WebKit has a GnuTLS-based HTTPS back end, among others.

IV. Certificate Validation in SSL/TLS

The only mechanism for server authentication in SSL/TLS is the client's validation of the server's X.509 public-key certificate presented during the handshake protocol. Client authentication is less common (in a typical HTTPS browsing session, only the server is authenticated). It involves symmetric steps on the server side to validate the client's certificate.

X.509 certificate validation is an extremely complex procedure, described in several semi-formal RFCs [64, 65, 66, 67, 68, 69, 70, 71]. Below, we give a very brief, partial overview of some of the key steps.

Chain of trust verification

Each SSL/TLS client trusts a number of *certificate authorities* (CAs), whose X.509 certificates are stored in the client's local "root of trust." We will refer to these trusted certificate authorities as *root CAs*, and to their certificates as *root certificates*. The list of root CAs varies from application to application and from OS to OS. For example, the Firefox Web browser ships with 144 root certificates pre-installed, while the Chrome Web browser on Linux and MacOS relies on the OS's list of root certificates.

Each X.509 certificate has an "issuer" field that contains the name of the certificate authority (CA) that issued the certificate. The certificate presented by the server (we'll call it the *leaf certificate*) should be accompanied by the certificate of the issuing CA and, if the issuing CA is not a root CA, the certificates of higher-level CAs all the way to a root CA.

As part of certificate validation, the client must construct a valid chain of certificates starting from the leaf certificate and ending in a root certificate (see an example in Fig. 1). Below,

we list some of the checks involved in validating the chain. These brief synopses are very informal and incomplete, please refer to RFC 5280 [69] for the full explanation.

Each certificate in the chain must be signed by the CA immediately above it and the root ("anchor") of the chain must be one of the client's trusted root CAs.

The current time must be later than the value of each certificate's "not valid before" field and earlier than the value of each certificate's "not valid after" field, in the time zone specified in these fields. If no time zone is specified, then Greenwich Mean Time (GMT) should be used.

If a CA certificate in an X.509 version 1 or version 2 certificate, then the client must either verify that it is indeed a CA certificate through out-of-band means or reject the certificate [69, 6.1.4(k)]. The following checks apply only to X.509 version 3 certificates.

For each CA certificate in the chain, the client must verify the *basic constraints* extension:

- The "CA bit" must be set. If the CA bit is not set, then the current certificate cannot act as a root or intermediate certificate in a certificate chain. The chain is not valid.
- If the CA certificate contains a "path length" constraint, the number of intermediate CAs between the leaf certificate and the current certificate must be less than the path length. For example, if the CA certificate has path length of 0, it can be used only to issue leaf certificates.

Every extension in a certificate is designated as *critical* or non-critical. A certificate with a critical extension that the client does not recognize or understand must be rejected.

If a CA certificate in the chain contains a *name constraints* extension, then the subject name in the immediately following certificate in the chain must satisfy the listed name constraints. Name constraints are used to limit the subjects that a CA can issue certificates for, by listing permitted or excluded subjects. This extension is critical.

If a certificate in the chain contains a *key usage* extension, the value of this extension must include the purpose that the certificate is being used for. For example, the key usage of an intermediate certificate must include keyCertSign (it must also have the CA bit set in the basic constraints, as described above). If a leaf certificate contains the server's RSA public key that will be used to encrypt a session key, its key usage extension must include keyEncipherment. CAs should mark this extension as critical.

Similar to key usage, if a certificate contains an *extended key usage* extension, the value of this extension must include the purpose that the certificate is being used for, e.g., server authentication in the case of a leaf certificate.

If a certificate contains an Authority Key Identifier (AKI) extension, then its value—containing the key identifier and/or issuer and serial number—should be used to locate the public key for validating the certificate. This extension is used when the certificate issuer has multiple public keys.

If a certificate contains a Certificate Revocation List (CRL) distribution points extension, the client should obtain CRL information as specified by this extension.

The above list omits many important checks and subtleties of certificate validation. For example, CA certificates may contain *policy constraints* that limit their authority in various ways [69, 4.2.1.11]. Policy constraints extension should be marked as critical, although in practice few SSL/TLS implementations understand policy constraints.

Hostname verification

After the chain of trust has been validated, the client must verify the server's identity by checking if the fully qualified DNS name of the server it wants to talk to matches one of the names in the "SubjectAltNames" extension or the "Common Name" field of the leaf certificate. Some SS-L/TLS implementations perform hostname verification, while others delegate it to higher-level applications (see Table IX).

V. Current testing practices for SSL/TLS implementations

Most SSL/TLS implementations analyzed in this paper ship with several pre-generated X.509 certificates intended for testing (Table I). These certificates differ only in a few fields, such as hashing algorithms (SHA-1, MD5, etc.), algorithms for public-key cryptography (DSA, RSA, Diffie-Hellman, etc.), and the sizes of public keys (512 bits, 1024 bits, etc.). OpenSSL uses a total of 2 certificates to test client and server authentication, respectively; the rest are intended to test other functionalities such as certificate parsing.

Testing with a handful of valid certificates is unlikely to uncover vulnerabilities, omissions, and implementation flaws in the certificate validation logic. For example, we found that GnuTLS mistakenly accepts all versions 1 certificates even though the default flag is set to accept only locally trusted version 1 root certificates (see Section IX). This vulnerability would have never been discovered with their existing test suite because it only contains version 3 certificates.

Automated adversarial testing is rarely, if ever, performed for SSL/TLS implementations. As we demonstrate in this paper, systematic testing with inputs that do *not* satisfy the protocol specification significantly improves the chances of uncovering subtle implementation flaws.

Several of the SSL/TLS implementations in our study, including OpenSSL, NSS, and MatrixSSL, have been tested and certified according to FIPS 140-2 [30], the U.S. government computer security standard for cryptographic modules. As the results of our testing demonstrate, FIPS certification does not mean that an implementation performs authentication correctly or is secure against man-in-the-middle attacks.

VI. Collecting Certificates

We used ZMap [26] to scan the Internet and attempt an SSL connection to every host listening on port 443. If the connection was successful, the certificate presented by the server was saved along with the IP of the host.

This scan yielded a corpus of 243,246 unique certificates. 23.5% of the collected certificates were already expired at the time they were presented by their servers, and 0.02% were not yet valid. The certificates in our corpus were issued by 33,837 unique issuers, identified by the value of their CN ("Common Name") field. Table II shows the 20 most common issuers.

23,698 of the certificates are X.509 version 1 (v1) certificates, 4,974 of which are expired. This is important because—as our testing has uncovered—any v1 certificate issued by a trusted CA can be used for man-in-the-middle attacks against several SSL/TLS implementations (see Section IX).

20,391 v1 certificates are self-signed. Table III shows the 10 most common issuers of the other 3,307 certificates. localhost, localdomain, and 192.168.1.1 are all self-issued certificate chains, but many v1 certificates have been issued by trusted issuers, especially manufacturers of embedded devices. For example, Remotewd.com is used for remote control of Western Digital Smart TVs, while UBNT and ZTE make networking equipment. As we show in Section IX, SSL/TLS implementations that specifically target embedded devices handle v1 certificates incorrectly and are thus vulnerable to man-in-the-middle attacks using these certificates.

437 certificates in our corpus have version 4, even though there is no X.509 version 4. 434 of them are self-signed, the other 3 are issued by Cyberoam, a manufacturer of hardware "security appliances." We conjecture that the cause is an off-by-one bug in the certificate issuance software: the version field in the certificate is zero-indexed, and if set to 3 by the issuer, it is interpreted as version 4 by SSL/TLS implementations.

Table IV shows the number of times various extensions show up in our corpus and how many unique values we observed for each extension. Extensions are labeled by short names if known, otherwise by their object identifiers (OID).

VII. Generating Frankencerts

The key challenge in generating test inputs for SSL/TLS implementations is how to create strings that (1) are parsed as X.509 certificates by the implementations, but (2) exercise parts of their functionality that are rarely or never executed when processing normal certificates.

We use our corpus of real certificates (see Section VI) as the source of syntactically valid certificate parts. Our algorithm them assembles these parts into random combinations we call *frankencerts*. One limitation of the certificates in our corpus is that they all conform to the X.509 specification. To test how SSL/TLS implementations behave when faced with syntactically valid certificates that do *not* conform to X.509, we also synthesize artificial certificate parts and add them to the inputs of the frankencerts generator (see Section VII-B).

A. Generating frankencerts

Algorithm 1 describes the generation of a single frankencert. Our prototype implementation of Frankencert is based on OpenSSL. It uses parts randomly selected from the corpus, with two exceptions: it generates a new RSA key and changes the issuer so that it can create chains where the generated frankencert acts as an intermediate certificate. The issuer field of

each frankencert must be equal to the subject of the certificate one level higher in the chain, or else all tested implementations fail to follow the chain and do not attempt to validate any other part of the certificate. For every other field, the generator picks the value from a randomly chosen certificate in the corpus (a different certificate for each field).

Extensions are set as follows. The generator chooses a random number of extensions from among all extensions observed in the corpus (Table IV). For each extension, it randomly chooses a value from the set of all observed values for that extension. Each value, no matter how common or rare, has an equal probability of appearing in a frankencert.

We use two CAs as roots of trust, with an X.509 version 1 certificate and an X.509 version 3 certificate, respectively. For the purposes of testing, both root CAs are installed in the local root of trust and thus trusted by all tested SSL/TLS clients.

Each frankencert is a well-formed X.509 certificate signed by a locally trusted CA, but it may be invalid for a number of reasons. By design, the frankencert generator does not respect the constraints on X.509 extensions. It also randomly designates extensions as critical or non-critical in each generated frankencert, violating the requirement that certain extensions must be critical (Section IV). This allows us to test whether SSL/TLS implementations reject certificates with unknown critical extensions, as required by the X.509 RFC [69].

For certificate chains, we use between 0 and 3 frankencerts. Each intermediate certificate uses the previous certificate's (randomly chosen) subject as its issuer and is signed by the previous certificate, creating a chain that SSL/TLS implementations can follow. These chains are well-formed, but may still be invalid because of the contents of random frankencerts acting as intermediate certificates. For example, the key usage extension of an intermediate certificate may not include keyCertSign, as required by the X.509 RFC [69], or an intermediate certificate may violate a name constraint which limits the set of subjects it is allowed to certify.

Algorithm 1

Generating a single frankencert

```
1:  procedure Frankencert(certs, exts, issuer)
2:      new_cert ← Create a blank cert
3:      for all field ∈ new_cert do
4:          if field ="key" then
5:              new_cert.key ← Create a random key
6:          else if field ="issuer" then
7:              new_cert.issuer ← issuer
8:          else
9:              random_cert ← CHOICE(certs)
10:             new_cert.field ← random_cert.field
11:         end if
```

```
12:     end for
13:     num_exts ← RANDOM(0, 10)
14:     for i ∈ 1..num_exts do
15:         random_id ← CHOICE(exts)
16:         random_val ← CHOICE(exts[random_id])
17:         new_cert.extensions[i].id ← random_id
18:         new_cert.extensions[i].val ← random_val
19:         if RANDOM < 0.05 then
20:             Flip(new_cert.extensions[i].critical)
21:         end if
22:     end for
23:     Sign(new_cert, issuer.key)
24:     return new_cert
25: end procedure
```

Algorithm 2

Generating a chain of frankencerts

```
1:  procedure Frankenchain(certs, ca, length)
2:      issuer ← ca
3:      chain ← Ø
4:      exts ← Getextensions(certs)
5:      for i ∈ 1..length do
6:          chain[i] ←Frankencert(certs, exts, issuer)
7:          issuer ← chain[i]
8:      end for
9:      return chain
10: end procedure
```

B. Generating synthetic mutations

The purpose of synthetic certificate parts is to test how SSL/TLS implementations react to extension values that follow the ASN.1 grammar for X.509 but do not conform to the X.509 specification.

Taking a frankencert as input, we first parse all extensions present in the certificate using OpenSSL. The critical bit and the rest of the extension value are extracted using X509_EXTENSION_get_critical() and X509_EXTENSION_get_data(), respectively. Then, for each of these extensions, the extension value is replaced with a randomly generated ASN.1 string and a null character (0) is probabilistically injected into this string. Because most of the SSL/TLS implementations in our testing are written in C, and C strings are terminated by a null character, this step helps verify whether implementations parse extension values correctly. Finally, the extension is randomly marked as critical or non-critical.

Algorithm 3

Extracting unique extensions from a corpus of certificates

```
1:  procedure GetExtensions(certs)
2:      uniq_exts ← Ø
3:      for all cert ∈ certs do
4:          for all ext ∈ cert.extensions do
5:              id ← ext.id
6:              val ← ext.val
7:              if id ∉ uniq_exts then
8:                  uniq_exts[id] ← Ø
9:              end if
10:             if val ∉ uniq_exts[id] then
11:                 uniq_exts[id] ← uniq_exts[id] ∪ val
12:             end if
13:         end for
14:     end for
15:     return uniq_exts
16: end procedure
```

VIII. Testing SSL/TLS Implementations

We tested open-source SSL/TLS libraries and several Web browsers. The tested libraries are OpenSSL 1.0.1e, PolarSSL 1.2.8, GnuTLS 3.1.9.1, CyaSSL 2.7.0, MatrixSSL 3.4.2, NSS 3.15.2, cryptlib 3.4.0-r1, OpenJDK 1.7.0_09-b30, and Bouncy Castle 1.49. The tested browsers are Firefox 20.0, Chrome 30.0.1599.114_p1, WebKitGTK 1.10.2-r300, Opera 12.0, Safari 7.0, and IE 10.0.

Testing was done in parallel on 3 machines: an Ubuntu Linux machine with two Intel Xeon E5420 (2.5Ghz) CPUs and 16 GB of RAM, an Ubuntu Linux machine with an Intel i7-2600K (4.0Ghz) CPU and 16GB of RAM, and a Gentoo Linux machine with an Intel i5-3360M (2.8Ghz) CPU with 8GB of RAM. Each machine generated and tested frankencerts independently, with the results merged later. The average speed of generating a frankencert chain with 3 certificates is 11.7ms.

SSL/TLS clients

We implemented a simple client for each SSL/TLS library. 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 server presents a frankencert. The client records the answer reported by the library, including error codes if any. When implementing these clients, we used the documentation provided by the libraries and followed the sample code in the documentation as closely as possible. We expect that most application developers using the library would follow the same procedure.

For testing Web browsers, we created scripts with the same input/output format as our clients for the libraries, allowing straightforward integration of browsers into our testing framework. For Firefox, we used Xulrunner to make an SSL connection and print the output without bringing up a Firefox window. For Chrome, we could not find an easy way to avoid launching the window. Therefore, we used a JavaScript file to make the connection and record the results.

Each execution of a library client takes between 0.04 and 0.10 seconds, with OpenSSL being the fastest and PolarSSL the slowest. The browser scripts are much slower: 0.6–1.0 seconds for Firefox and 1.1–1.4 seconds for Chrome.

Differential testing

For differential testing of multiple SS-L/TLS implementations, we implemented a Python script that generates frankencerts and executes all clients against each frankencert. The entire script is 367 lines of code, including 102 lines for certificate generation and 163 lines for parallel execution of clients. Certificates are generated in batches of 200; executing all clients on a single batch takes 25 seconds.

If a certificate causes disagreement between the clients (i.e., the clients produce different error codes when presented with this certificate), the certificate is indexed by its SHA-1 hash and stored into the appropriate bucket. Buckets are defined by the tuples of error codes returned by each client. For example, if client A accepts the certificate, client B rejects it with error code 34, and client C rejects it with error code 1, the certificate is stored into the 0-34-1 bucket. The size of each bucket is capped at 512 certificates.

In total, we tested our clients on **8,127,600 frankencerts**. It is not computationally feasible to exhaustively generate certificates with all possible combinations of extension values from Table IV, but every value of every extension appeared in at least one of the frankencerts used in the testing.

Our testing yielded **208 distinct discrepancies between SSL/TLS implementations**, with a total of 62,022 frankencerts triggering these discrepancies.

Analysis of the results

All SSL/TLS implementations we tested are supposed to implement the same protocol and, in particular, exactly the same certificate validation logic. Whenever one implementation accepts a certificate and another rejects the same certificate, their implementations of the X.509 standard must be semantically different. In other words, differential testing has no **false positives**. This is very important when testing on over 8 million inputs, because any non-negligible false-positive rate would have resulted in an overwhelming number of false positives.

While all discrepancies found by differential testing indicate genuine differences between implementations, not every difference implies a security vulnerability. For each discrepancy, we manually analyzed the source code of the disagreeing implementations to identify the root cause of the disagreement and find the flaw (if any) in the certificate validation logic of

one or more implementations. Because some parts of the X.509 standard are left to the discretion of the implementation, a few of the discrepancies turned out to be benign. For example, the differing treatments of the Authority Key Identifier (AKI) extension (Section IX-E) fall into this category.

Differential testing with frankencerts suffers from **false negatives** and can miss security flaws. SSL/TLS implementations may contain code paths that are not exercised by a given set of frankencerts. An example of this is the recently discovered certificate validation bug in GnuTLS [33], which is only triggered by syntactically malformed certificates. It was not found by our testing because all frankencerts we generated comply with the X.509 grammar. Similarly, frankencerts will not trigger flaws on the code paths responsible for processing extensions that do not occur in the certificate corpus from which these frankencerts are constructed, or the paths executed only for certain versions and modes of SSL/TLS, etc.

Further, if all implementations make the same mistake, it will not manifest as a discrepancy. Finally, an implementation may reject an invalid certificate for the wrong reason(s). To reduce false negatives in the latter case, we also analyzed the discrepancies between the reported validation errors.

Analysis of error reporting

Proper error reporting is critical for SSL/TLS implementations because a trivial, low-risk warning (e.g., expired certificate) may accidentally hide or mask a severe problem (e.g., invalid certificate issuer).

Not every SSL/TLS implementation produces fine-grained error codes that are easy to translate into a human-understandable reason for rejection. Many simply reject the certificate and return a generic error. If the certificate is invalid for multiple reasons, all libraries except GnuTLS return only one error value, but some allow the application to extract more error codes through additional function calls. This is fraught with peril because the application may forget to make these additional calls and thus allow a less severe error to mask a serious problem with the certificate.

Therefore, we limited our differential testing of error reporting to Web browsers, OpenSSL, NSS, GnuTLS, and OpenJDK. For this testing, each output was mapped to one of the following reasons: “Accepted,” “Invalid issuer,” “Expired,” “Not yet valid,” and “Unknown or invalid critical extension.” For Web browsers, we also included “Hostname in the certificate does not match the server.”

IX. Results

Depending on the combination of mutations in a frankencert, the same flaw in a given implementation of X.509 certificate validation can produce different results. We analyzed 208 discrepancies between the implementations found by our testing and attributed them to 15 distinct root causes.

Table V summarizes the results. As the second column shows, most of the issues could not have been discovered without frankencerts because the certificates triggering these issues do

not exist in our corpus (but, of course, can be crafted by the adversary to exploit the corresponding flaw).

A. Incorrect checking of basic constraints

Basic constraints, described in Section IV, are an essential part of CA certificates. Every X.509 version 3 CA certificate must have the CA bit set, otherwise any domain with a valid leaf certificate could act as a rogue CA and issue fake certificates for other domains.

Untrusted version 1 intermediate certificate—Before version 3, X.509 certificates did not have basic constraints, making it impossible to check whether a certificate in the chain belongs to a valid CA except via out-of-band means. If an SSL/TLS implementation encounters a version 1 (v1) CA certificate that cannot be validated out of band, it must reject it [69, 6.1.4(k)].

Both MatrixSSL and GnuTLS accept chains containing v1 certificates. As we explain below, this can make any application based on MatrixSSL or GnuTLS vulnerable to man-in-the-middle attacks. In MatrixSSL, the following code silently skips the basic constraints check for any certificate whose version field is 0 or 1 (encoding X.509 version 1 or 2, respectively, because the version field is zero-indexed):

```
/* Certificate authority constraint only available in
version 3 certs */
if ((ic->version > 1) && (ic->extensions.bc.ca<= 0)) {
psTraceCrypto("Issuer does not have basicConstraint
CA permissions\n");
sc->authStatus = PS_CERT_AUTH_FAIL_BC;
return PS_CERT_AUTH_FAIL_BC;
}
```

GnuTLS, on the other hand, contains a very subtle error. This error could not have been uncovered without frankencerts because none of the real certificate chains in our corpus contain v1 intermediate certificates.

GnuTLS has three flags that an application can set to customize the library's treatment of v1 CA certificates: GNUTLS_VERIFY_ALLOW_X509_V1_CA_CRT (only accept v1 root certificates), GNUTLS_VERIFY_ALLOW_ANY_X509_V1_CA_CRT (accept v1 certificates for root and intermediate CAs), and GNUTLS_VERIFY_DO_NOT_ALLOW_X509_V1_CA_CRT (reject all v1 CA certificates). Only GNUTLS_VERIFY_ALLOW_X509_V1_CA_CRT is set by default. The intention is good: the application may locally trust a v1 root CA, but, to prevent other customers of that root CA from acting as CAs themselves, no v1 intermediate certificates should be accepted.

The relevant part of GnuTLS certificate validation code is shown below (adapted from lib/x509/verify.c). After a root v1 certificate has been accepted, GnuTLS needs to prevent any

further v1 certificates from being accepted. To this end, it clears the GNUTLS_VERIFY_ALLOW_X509_V1_CA_CRT flag on line 12 before calling _gnutls_verify_certificate2. The latter function accepts v1 certificates unless a different flag, GNUTLS_VERIFY_DO_NOT_ALLOW_X509_V1_CA_CRT is set (line 25).

```
1 unsigned int _gnutls_x509_verify_certificate( . . . )
2 {
3 . . .
4
5 /* verify the certificate path (chain) */
6 for (i = clist_size - 1; i > 0; i--)
7 {
8 /* note that here we disable this V1 CA flag. So
that no version 1
9 * certificates can exist in a supplied chain.
10 */
11 if (!(flags &
GNUTLS_VERIFY_ALLOW_ANY_X509_V1_CA_CRT))
12 flags &= ~(GNUTLS_VERIFY_ALLOW_X509_V1_CA_CRT);
13 if ((ret = _gnutls_verify_certificate2 ( . . . )) ==
0)
14 {
15 /* return error */
16 }
17 }
18 . . .
19 }
20
21 int _gnutls_verify_certificate2( . . . )
22 {
23 . . .
24 if (!(flags & GNUTLS_VERIFY_DISABLE_CA_SIGN) &&
25 ((flags &
GNUTLS_VERIFY_DO_NOT_ALLOW_X509_V1_CA_CRT)
26 || issuer_version != 1))
27 {
28 if (check_if_ca (cert, issuer, flags) == 0)
29 {
30 /*return error*/
31 . . .
32 }
33 }
34 /*perform other checks*/
```

```
35 . . .
36 }
```

There is an interesting dependency between the two flags. To prevent intermediate v1 certificates from being accepted, GNUTLS_VERIFY_ALLOW_X509_V1_CA_CRT must be false *and* GNUTLS_VERIFY_DO_NOT_ALLOW_X509_V1_CA_CRT must be true. The calling function sets the former, but not the latter. Therefore, although by default GnuTLS is only intended to accept root v1 certificates, in reality it accepts *any* v1 certificate.

The consequences of this bug are not subtle. If an application based on GnuTLS trusts a v1 root CA certificate, then **any server certified by the same root can act as a rogue CA**, issuing fake certificates for any Internet domain and launching man-in-the-middle attacks against this GnuTLS-based application. Unfortunately, trusting v1 root certificates is very common. For example, Gentoo Linux by default has 13 v1 root CA certificates, Mozilla has 9, and we observed thousands of CA-issued v1 leaf certificates “in the wild” (Section VI).

Untrusted version 2 intermediate certificate—We never observed X.509 version 2 certificates “in the wild,” but, for the purposes of testing, did generate version 2 frankencerts.

As explained above, MatrixSSL silently accepts all CA certificates whose version field is less than 2 (i.e., version number less than 3). In GnuTLS, gnutls_x509_crt_get_version returns the actual version, not the version field, and the following check blocks version 2 certificates:

```
issuer_version = gnutls_x509_crt_get_version (issuer);
// ...
if (!(flags & GNUTLS_VERIFY_DISABLE_CA_SIGN) &&
((flags &
GNUTLS_VERIFY_DO_NOT_ALLOW_X509_V1_CA_CRT)
|| issuer_version != 1))
{
// ...
}
```

Version 1 certificate with valid basic constraints—Basic constraints were added only in X.509 version 3, but several SSL/TLS implementations always verify basic constraints if present in the certificate regardless of its version field.

Some of our frankencert chains include version 1 intermediate certificates with correct basic constraints (obviously, such certificates do not exist “in the wild”). OpenSSL, GnuTLS, MatrixSSL, CyaSSL, Opera, and Chrome accept them, Open-JDK and Bouncy Castle reject them, NSS and Firefox fail with a generic *Security library failure* error. Neither choice appears to lead to a security vulnerability.

Intermediate CA not authorized to issue further intermediate CA certificates— When a higher-level CA certifies a lower-level CA, it can impose various restrictions on the latter. For example, it can limit the number of intermediate certificates that may follow the lower-level CA's certificate in a certificate chain. This is done by setting the pathLenConstraint field in the basic constraints extension of the lower-level CA's certificate.

For example, if path length is set to zero, then the lower-level CA is authorized to issue only leaf certificates, but not intermediate CA certificates. This is good security practice: a CA delegates its authority to a lower-level CA, but prevents the latter from delegating it any further. We observed 17 CA certificates with path length constraints in our corpus.

MatrixSSL ignores path length constraints. This can be exploited by a malicious or compromised CA to evade restrictions imposed by a higher-level CA. For example, suppose that a trusted root CA authorized a lower-level CA—call it EnterpriseCA—but prohibited it from creating other CAs (via path length constraints) and from issuing certificates for any domain other than enterprise.com (via name constraints—see Section IX-B). This provides a degree of protection if EnterpriseCA is compromised. If the attacker uses EnterpriseCA to issue a certificate for, say, google.com, this certificate should be rejected by any SSL/TLS implementation because it violates the constraints expressed in EnterpriseCA's own certificate.

This attack will succeed, however, against any application based on MatrixSSL. The impact of this vulnerability may be amplified by the fact that MatrixSSL targets embedded devices, whose manufacturers are the kind of organizations that are likely to obtain CA certificates with restricted authority.

There is an interesting discrepancy in how the implementations react when an intermediate CA whose path length is zero is followed by a leaf certificate that also happens to be a CA certificate. In our testing, only MatrixSSL and GnuTLS accepted this chain. All other SSL/TLS implementations rejected it because they do not allow any CA certificate to follow an intermediate CA whose path length is zero. This interpretation is incorrect. The X.509 standard explicitly permits a leaf CA certificate to follow an intermediate CA whose path length is zero [69, Section 4.2.1.9], but only GnuTLS implements this part of the standard correctly.

B. Incorrect checking of name constraints

The higher-level CA may restrict the ability of a lower-level CA to issue certificates for arbitrary domains by including a name constraint in the lower-level's CA's certificate. For example, if the issuing CA wants to allow the lower-level CA to certify only the subdomains of foo.com, it can add a name constraint *.foo.com to the lower-level CA's certificate.

GnuTLS, MatrixSSL, and CyaSSL ignore name constraints and accept the server's certificate even if it has been issued by a CA that is not authorized to issue certificates for that server.

C. Incorrect checking of time

Every X.509 certificate has the *notBefore* and *notAfter* timestamp fields. The SSL/TLS client must verify that the current date and time in GMT (or the time zone specified in these fields) is within the range of these timestamps.

PolarSSL ignores the *notBefore* timestamp and thus accepts certificates that are not yet valid. When verifying the *notAfter* field, it uses local time instead of GMT or the time zone specified in the field.

MatrixSSL does not perform any time checks of its own and delegates this responsibility to the applications. The sample application code included with MatrixSSL checks the day, but not the hours and minutes of the *notAfter* field, and uses local time, not GMT or the time zone specified in the field.

D. Incorrect checking of key usage

SSL/TLS clients must check the key usage and, if present, extended key usage extensions to verify that the certificates are authorized for their purpose. Leaf certificates must be authorized for key encipherment or key agreement, while CA certificates must be authorized to sign other certificates.

CA certificate not authorized for signing other certificates—All CA certificates in the chain must include keyCertSign in their key usage. GnuTLS, CyaSSL, and MatrixSSL do not check the key usage extension in CA certificates. An attacker who compromises any CA key, even a key that is not intended or used for certificate issuance, can use it to forge certificates and launch man-in-the-middle attacks against applications based on these libraries.

Server certificate not authorized for use in SSL/TLS handshake—PolarSSL, GnuTLS, CyaSSL, and MatrixSSL do not check the key usage extension in leaf certificates. This is a serious security vulnerability. For example, if an attacker compromises some company’s code signing certificate, which is only intended for authenticating code, he will be able to impersonate that company’s network and Web servers to any application based on the above SSL/TLS libraries, vastly increasing the impact of the attack.

Server certificate not authorized for server authentication—PolarSSL, gnuTLS, CyaSSL and MatrixSSL do not check the extended key usage extension. Given a certificate with key usage that allows all operations and extended key usage that only allows it to be used for TLS *client* authentication (or any purpose other than server authentication), these libraries accept the certificate for server authentication.

E. Other discrepancies in extension checks

Unknown critical extensions—If an SSL/TLS implementation does not recognize an extension that is marked as critical, it must reject the certificate. GnuTLS, CyaSSL, and MatrixSSL accept certificates with unknown critical extensions.

Malformed extension values—Given a certificate with a known non-critical extension whose value is syntactically well-formed ASN.1 but not a valid value for that extension, OpenSSL, GnuTLS, CyaSSL, and MatrixSSL accept it, while the other libraries and all browsers reject it.

Inconsistencies in the definition of self-signed—Self-issued certificates are CA certificates in which the issuer and subject are the same entity [69]. Nevertheless, given a (very odd) certificate whose subject is the same as issuer but that also has a valid chain of trust, GnuTLS and MatrixSSL accept it.

Inconsistencies between the certificate's Authority Key Identifier and its issuer—The Authority Key Identifier (AKI) extension differentiates between multiple certificates of the same issuer. When an AKI is present in a certificate issued by CA whose name is *A*, but the AKI points to a certificate whose subject name is *B*, some libraries reject, others accept.

If the serial number field is absent in the AKI, then GnuTLS accepts. But if this field is present and does not match the issuer's serial number, then GnuTLS rejects.

F. "Users… don't go for the commercial CA racket"

We planned to include cryptlib in our testing, but then discovered that it does not verify certificate chains. We let the following code snippet, taken from session/ssl_cli.c, speak for itself (there is no code inside the **if** block):

```
/* If certificate verification hasn't been disabled
, make sure that
the
server's certificate verifies */
if( !( verifyFlags & SSL_PFLAG_DISABLE_CERTVERIFY )
)
{
/* This is still too risky to enable by default
because most users outside of web browsing
don't go for the commercial CA racket */
}
return( CRYPT_OK );
```

G. Security problems in error reporting

Rejection of an invalid certificate is not the end of the story. Web browsers and other interactive applications generate a warning based on the reason for rejection, show this warning to the user, and, in many cases, allow the user to override the dialog and proceed.

Different errors have different security implications. A recently expired, but otherwise valid certificate may be evidence of a sloppy system administrator who forgot to install a new

certificate, but does not imply that the SSL/TLS connection will be insecure. “Expired certificate” warnings are sufficiently common that users have learned to ignore them and browser developers are even advised to suppress them [1].

If, on the other hand, the certificate issuer is not valid, this means that the server cannot be authenticated and the connection is not secure against man-in-the-middle attacks. If the server’s hostname does not match the subject of the certificate, the user may inspect both names and decide whether to proceed or not. For example, if the hostname (e.g., bar.foo.com) is a subdomain of the common name in the certificate (e.g., foo.com), the user may chalk the discrepancy up to a minor misconfiguration and proceed.

To test whether SSL/TLS implementations report certificate errors correctly, we performed differential testing on leaf certificates with all combinations of the following:

- **Expired (E)**: Current time is later than the *notAfter* timestamp in the certificate.
- **Bad issuer (I)**: There is no valid chain of trust from the certificate’s issuer to a trusted root CA.
- **Bad name (N)**: Neither the common name, nor the subject alternative name in the certificate match the server’s hostname.

I is the most severe error. It implies that the connection is insecure and *must* be reported to the user. On the other hand, *E* is a common, relatively low-risk error.

Table VI shows the results. For these tests, we extended our client suite with common Web browsers, since they are directly responsible for interpreting the reasons for certificate rejection and presenting error warnings to human users.

Most SSL/TLS implementations and Web browsers return only one error code even if the certificate is invalid for multiple reasons. What is especially worrisome is that some browsers choose to report the less severe reason. In effect, they **hide a severe security problem under a low-risk warning**. These cases are highlighted in bold in Table VI.

For example, if a network attacker—say, a malicious WiFi access point—presents a self-signed, very recently expired certificate for gmail.com or any other important domain to a user of Safari 7 or Chrome 30 (on Linux), the *only* error warning the user will see is “Expired certificate.” [1] Many users will click through this low-risk warning—even though authentication has failed and the server has been spoofed! This vulnerability is *generic* in all NSS-based applications: if the certificate is expired, that’s the only reported error code regardless of any other problems with the certificate.

A related problem (not reflected in Table VI) is caused by “Weak Key” warnings. When presented with a certificate containing a 512-bit RSA key, Firefox and Chrome accept it, while Opera warns that the key is weak. If the certificate is invalid, Opera still produces the same “Weak Key” warning, masking other problems with the certificate, e.g., invalid issuer.

[1]As this paper was being prepared, the same bug was reported in http://news.netcraft.com/archives/2013/10/16/us-government-aiding-spying-against-itself.html

The other warnings are available in the details tab of the error dialog, assuming Opera users know to look there.

Finally, if Firefox encounters two certs issued by the same CA that have the same serial number, it shows an error message describing the problem. This message masks all other warnings, but there is no way for the user to override it and proceed, so this behavior is safe.

H. Other checks

Weak cryptographic hash functions—Digital signatures on SSL/TLS certificates can use a variety of cryptographic hash (aka message digest) functions. As Table VII shows, only NSS, GnuTLS, and Chrome reject MD5 certificates, which are known to be vulnerable to prefix-collision attacks [77].

Short keys—Table VIII shows that virtually all tested implementations support short keys (512 bits for RSA) and unusual key sizes (1023 bits, chosen because it occurs 87 times in our certificate corpus).

Additional checks—Table IX summarizes which SSL/TLS libraries perform additional checks, such as Certification Revocation Lists (CRL), subject alternative name, and hostname. The latter check is critically important for security against man-in-the-middle attacks [31], but often delegated by libraries to higher-level applications.

X. Developer Responses

We notified the developers of all affected SSL/TLS implementations about the issues discovered by our testing.

GnuTLS has fixed the bug involving version 1 intermediate CA certificates (starting from version 3.2.11) and also created a patch for older versions. A security advisory (CVE-2014-1959) has been issued for this bug. GnuTLS used to check the keyUsage field in earlier versions, but removed these checks after getting bug reports from developers who were using certificates with incorrect keyUsage fields.[2] This was necessary for compatibility with several other SSL/TLS implementations that do not check this field. Delignat-Lavaud et al. [19] independently reported that GnuTLS does not reject certificates with unknown critical extensions. According to GnuTLS, rejecting such certificates may allow certain corporations to lock out GnuTLS by issuing certificates with custom extensions and thus forcing developers to use the corporation's own SSL library instead of GnuTLS.

MatrixSSL plans to reject version 1 intermediate CAs and check path length constraints starting from the next release. In general, MatrixSSL only performs basic checks on the certificate and depends on the application-provided callbacks to check key usage, extended key usage, expiration timestamps, etc. To facilitate these checks, MatrixSSL will parse the critical flags and the extended key usage extension. Since MatrixSSL primarily targets embedded devices, which do not always have the time zone information, in most cases the

[2] http://www.gnutls.org/faq.html

notBefore and *notAfter* timestamps in the certificate will have to be checked against the available local time.

CyaSSL is fixing all reported issues. The fixes will be part of CyaSSL 3.0.0, expected to be released in April 2014.

PolarSSL is currently working on the fixes.

cryptlib does not support certificate chain validation to avoid validation failures for the users who run their own CA hierarchy or do not use certificates. The cryptlib manual[3] recommends other techniques for authenticating the server, such as matching key fingerprints. In addition, it strongly recommends using the PSK cipher suites for mutual authentication of both the client and server. The manual also provides an outline for the application writers who want to use certificates on how to perform certificate validation on their own.

NSS developers informed us that all Mozilla products use a glue layer called Personal Security Manager (PSM) over NSS instead of using NSS directly. The PSM certificate validation routine, CERT_VerifyCertificate, takes an argument named CERTVerifyLog that, if not set to NULL, returns a list of all certificate validation errors. An example usage of the function can be found at http://mxr.mozilla.org/mozilla-central/source/security/manager/ssl/src/SSLServerCertVerification.cpp#622

As of this writing, we are still talking to Web-browser developers about user warnings generated by their browsers when certificate validation fails.

XI. Conclusions

We designed, implemented, and applied the first automated method for large-scale adversarial testing of certificate validation logic in SSL/TLS implementations. Our key technical innovation is “frankencerts,” synthetic certificates randomly mutated from parts of real certificates. Frankencerts are syntactically well-formed, but may violate the X.509 specification and thus exercise rarely tested functionality in SSL/TLS implementations. Our testing uncovered multiple flaws in popular SSL/TLS libraries and Web browsers, including security vulnerabilities that break server authentication guarantees and can be exploited for stealthy man-in-the-middle attacks.

Certificate validation is only one part of the SSL/TLS handshake. Bugs in other parts of the handshake—e.g., accidentally omitting to check that the server’s messages are signed with the key that matches the certificate [49]—and incorrect usage of SSL/TLS implementations by higher-level software [29, 31] can completely disable authentication and leave applications vulnerable to man-in-the-middle attacks. Development of automated methods that can analyze the entire SSL/TLS software stack and prove that it has been implemented securely and correctly remains an open challenge.

[3] http://www.cryptlib.com/downloads/manual.pdf, page 118

Acknowledgments

We are grateful to Rui Qiu for participating in the initial exploration of the ideas that led to this work, and to our Oakland shepherd Matthew Smith for helping smooth ruffled feathers. This work was partially supported by the NSF grants CNS-0746888, CCF-0845628, and CNS-1223396, a Google research award, NIH grant R01 LM011028-01 from the National Library of Medicine, and Google PhD Fellowship to Suman Jana.

References

1. Akhawe D, Amann B, Vallentin M, Sommer R. Here's my cert, so trust me, maybe? Understanding TLS errors on the Web. WWW. 2013
2. Akhawe D, Felt A. Alice in Warningland: A large-scale field study of browser security warning effectiveness. USENIX Security. 2013
3. AlFardan N, Paterson K. Lucky thirteen: Breaking the TLS and DTLS record protocols. S&P. 2013
4. Amann B, Sommer R, Vallentin M, Hall S. No attack necessary: The surprising dynamics of SSL trust relationships. ACSAC. 2013
5. Amrutkar C, Singh K, Verma A, Traynor P. VulnerableMe: Measuring systemic weaknesses in mobile browser security. ICISS. 2012
6. Amrutkar C, Traynor P, van Oorschot P. An empirical evaluation of security indicators in mobile Web browsers. IEEE Trans Mobile Computing. 2013
7. Anand S, Burke E, Chen T, Clark J, Cohen M, Grieskamp W, Harman M, Harrold M, McMinn P. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software. 2013; 86(8):1978–2001.
8. Bleichenbacher D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. CRYPTO. 1996
9. Brumley D, Boneh D. Remote timing attacks are practical. USENIX Security. 2003
10. Brumley D, Caballero J, Liang Z, Newsome J, Song D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. USENIX Security. 2007
11. Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI. 2008
12. Cadar C, Engler D. Execution generated test cases: How to make systems code crash itself. SPIN. 2005
13. Chandrasekhar B, Khurshid S, Marinov D. Korat: Automated testing based on Java predicates. ISSTA. 2002
14. Chen, T.; Cheung, S.; Yiu, S. Technical Report HKUST-CS98-01. Department of Computer Science, Hong Kong University of Science and Technology; 1998. Metamorphic testing: A new approach for generating next test cases.
15. Cheon Y, Leavens G. A simple and practical approach to unit testing: The JML and JUnit way. ECOOP. 2002
16. Clark J, van Oorschot P. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. S&P. 2013
17. Comodo report of incident. 2011. http://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html
18. Daniel B, Dig D, Garcia K, Marinov D. Automated testing of refactoring engines. FSE. 2007
19. Delignat-Lavaud A, Abadi M, Birrell A, Mironov I, Wobber T, Xie Y. Web PKI: Closing the gap between guidelines and practices. NDSS. 2014
20. Dickinson W, Leon D, Podgurski A. Finding failures by cluster analysis of execution profiles. ICSE. 2001
21. Dietz M, Czeskis A, Balfanz D, Wallach D. Origin-bound certificates: A fresh approach to strong client authentication for the Web. USENIX Security. 2012

22. Diginotar issues dodgy SSL certificates for Google services after break-in. 2011. http://www.theinquirer.net/inquirer/news/2105321/diginotar-issues-dodgy-ssl-certificates-google-services-break
23. Dijkstra E. A Discipline of Programming. 1976
24. Duong, T.; Rizzo, J. Here come the ninjas. 2011. http://nerdoholic.org/uploads/dergln/beastpart2/ssljun21.pdf
25. Durumeric Z, Kasten J, Bailey M, Halderman A. Analysis of the HTTPS certificate ecosystem. IMC. 2013
26. Durumeric Z, Wustrow E, Halderman A. ZMap: Fast Internet-wide scanning and its security applications. USENIX Security. 2013
27. Eckersley P, Burns J. An observatory for the SSLiverse. DEFCON. 2010
28. Ernst M. Static and dynamic analysis: Synergy and duality. WODA. 2003
29. Fahl S, Harbach M, Muders T, Smith M. Why Eve and Mallory love Android: An analysis of SSl (in)security on Android. CCS. 2012
30. FIPS PUB 140-2. Security requirements for cryptographic modules. 2001. http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
31. Georgiev M, Iyengar S, Jana S, Anubhai R, Boneh D, Shmatikov V. The most dangerous code in the world: Validating SSL certificates in non-browser software. CCS. 2012
32. Gligoric M, Behrang F, Li Y, Overbey J, Hafiz M, Marinov D. Systematic testing of refactoring engines on real software projects. ECOOP. 2013
33. CVE-2014-0092. 2014. https://bugzilla.redhat.com/showbug.cgi?id=1069865
34. Godefroid P, Kiezun A, Levin M. Grammar-based whitebox fuzzing. PLDI. 2008
35. Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. PLDI. 2005
36. Godefroid P, Levin M, Molnar D. Automated whitebox fuzz testing. NDSS. 2008
37. Halfond W, Anand S, Orso A. Precise interface identification to improve testing and analysis of web applications. ISSTA. 2009
38. Heninger N, Durumeric Z, Wustrow E, Halderman A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. USENIX Security. 2012
39. Holland, JH. Adaptation in Natural and Artificial Systems. 2. University of Michigan Press; 1992. p. 1975
40. CVE-2011-0228. 2011. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0228
41. Jagannath V, Lee Y, Daniel B, Marinov D. Reducing the costs of bounded-exhaustive testing. FASE. 2009
42. Jana S, Shmatikov V. Abusing file processing in malware detectors for fun and profit. S&P. 2012
43. Jones J, Bowring J, Harrold M. Debugging in parallel. ISSTA. 2007
44. Kaminsky D, Patterson M, Sassaman L. PKI layer cake: New collision attacks against the global X.509 infrastructure. FC. 2010
45. Khurshid S, Pasareanu C, Visser W. Generalized symbolic execution for model checking and testing. TACAS. 2003
46. Kiezun A, Guo P, Jayaraman K, Ernst M. Automatic creation of SQL injection and cross-site scripting attacks. ICSE. 2009
47. King J. Symbolic execution and program testing. Commun ACM. 1976; 19(7)
48. Lammel, R.; Schulte, W. Testing of Communicating Systems. Lecture Notes in Computer Science; 2006. Controllable combinatorial coverage in grammar-based testing; p. 19-38.
49. Langley, A. Apple's SSL/TLS bug. 2014. https://www.imperialviolet.org/2014/02/22/applebug.html
50. Lenstra, A.; Hughes, J.; Augier, M.; Bos, J.; Kleinjung, T.; Wachter, C. Ron was wrong, Whit is right. 2012. http://eprint.iacr.org/2012/064
51. Majumdar R, Xu R. Directed test generation using symbolic grammars. ASE. 2007
52. Malloy B, Power J. An interpretation of Purdom's algorithm for automatic generation of test cases. ICIS. 2001

53. Marinov D, Khurshid S. TestEra: A novel framework for automated testing of Java programs. ASE. 2001
54. Marlinspike, M. IE SSL vulnerability. 2002. http://www.thoughtcrime.org/ie-ssl-chain.txt
55. Marlinspike, M. More tricks for defeating SSL in practice. DEFCON; 2009.
56. Marlinspike, M. New tricks for defeating SSL in practice. Black Hat DC; 2009.
57. Marlinspike, M. Null prefix attacks against SSL/TLS certificates. 2009. http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf
58. Maurer P. Generating test data with enhanced context-free grammars. IEEE Software. 1990; 7(4): 50–55.
59. McKeeman W. Differential testing for software. Digital Technical Journal. 1998; 10(1):100–107.
60. Parsovs A. Practical issues with TLS client certificate authentication. NDSS. 2014
61. Podgurski A, Leon D, Francis P, Masri W, Minch M, Sun J, Wang B. Automated support for classifying software failure reports. ICSE. 2003
62. Purdom P. A sentence generator for testing parsers. BIT Numerical Mathematics. 1972; 12:366–375.
63. Ramos D, Engler D. Practical, low-effort equivalence verification of real code. CAV. 2011
64. The TLS protocol version 1.0. 1999. http://tools.ietf.org/html/rfc2246
65. Internet X.509 public key infrastructure certificate policy and certification practices framework. 1999. http://www.ietf.org/rfc/rfc2527.txt
66. HTTP over TLS. 2000. http://www.ietf.org/rfc/rfc2818.txt
67. The Transport Layer Security (TLS) protocol version 1.1. 2006. http://tools.ietf.org/html/rfc4346
68. The Transport Layer Security (TLS) protocol version 1.2. 2008. http://tools.ietf.org/html/rfc5246
69. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. 2008. http://tools.ietf.org/html/rfc5280
70. The Secure Sockets Layer (SSL) protocol version 3.0. 2011. http://tools.ietf.org/html/rfc6101
71. 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). 2011. http://tools.ietf.org/html/rfc6125
72. Rizzo J, Duong T. The CRIME attack. Ekoparty. 2012
73. Saxena P, Akhawe D, Hanna S, Mao F, McCamant S, Song D. A symbolic execution framework for JavaScript. S&P. 2010
74. Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C. FSE. 2005
75. Sirer, E.; Bershad, B. Using production grammars in software testing. Proc. 2nd Conference on Domain-specific Languages; 1999;
76. Srivastava V, Bond M, McKinley K, Shmatikov V. A security policy oracle: Detecting security holes using multiple API implementations. PLDI. 2011
77. Stevens M, Sotirov A, Appelbaum J, Lenstra A, Molnar D, Osvik D, Weger B. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. CRYPTO. 2009
78. Vratonjic N, Freudiger J, Bindschaedler V, Hubaux J-P. The inconvenient truth about Web certificates. WEIS. 2011
79. Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. PLDI. 2011

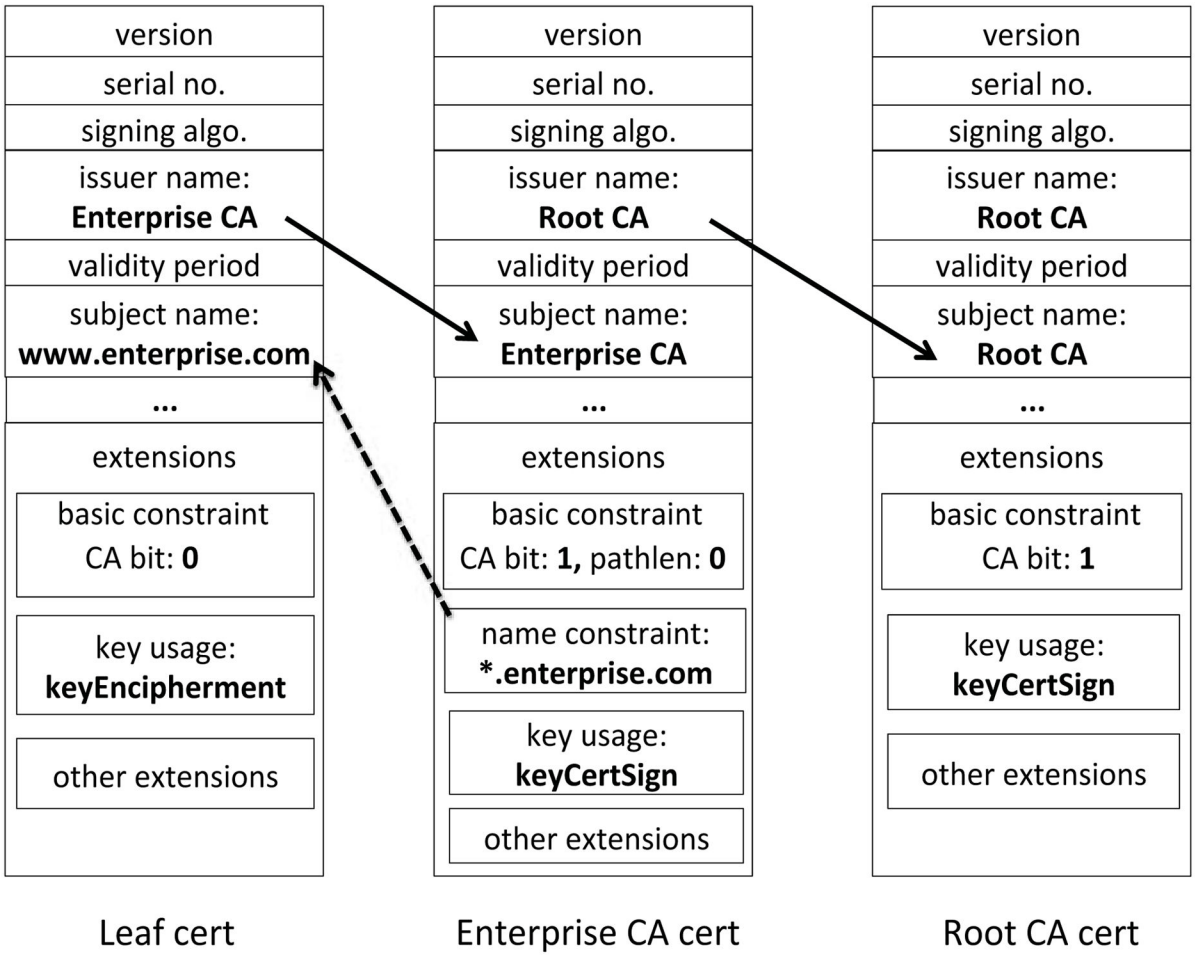

Fig. 1.
A sample X509 certificate chain.

TABLE I

Number of SSL/TLS certificates used by different implementations for testing

Implementation	Certificate count
NSS	64
GnuTLS	51
OpenSSL	44
PolarSSL	18
CyaSSL	9
MatrixSSL	9

TABLE II

20 most common issuers in our corpus

Common Name (CN)	Occurrences
Cybertrust Public SureServer SV CA	30066
Go Daddy Secure Certification Authority	13300
localhost.localdomain	7179
GeoTrust SSL CA	7171
COMODO SSL CA	7114
RapidSSL CA	6358
COMODO SSL CA 2	5326
BMS	4878
DigiCert High Assurance CA-3	4341
Hitron Technologies Cable Modem Root Certificate Authority	4013
VeriSign Class 3 Secure Server CA - G3	3837
COMODO High-Assurance Secure Server CA	3681
PositiveSSL CA 2	2724
Entrust Certification Authority - L1C	2719
Daniel	2639
Vodafone (Secure Networks)	2634
192.168.168.168	2417
GeoTrust DV SSL CA	2174
localhost	2142
Parallels Panel	2084

TABLE III

10 most common issuers of X.509 version 1 certificates

Common Name (CN)	Occurrences
BMS	4877
Parallels Panel	2003
localhost	1668
brutus.neuronio.pt	1196
plesk	1163
remotewd.com	1120
UBNT	1094
localdomain	986
192.168.1.1	507
ZTE Corporation	501

TABLE IV

Extensions observed in our corpus

Name or OID	Occurrences	Unique values
basicConstraints	161723	13
authorityKeyIdentifier	161572	21990
subjectKeyIdentifier	151823	72496
keyUsage	132970	54
extendedKeyUsage	131453	83
crlDistributionPoints	126579	4851
subjectAltName	101622	59767
authorityInfoAccess	89005	3864
certificatePolicies	81264	418
nsCertType	63913	21
nsComment	5870	185
1.3.6.1.4.1.311.20.2	2897	11
issuerAltName	1519	115
1.3.6.1.5.5.7.1.12	1474	2
SMIME-CAPS	915	4
1.3.6.1.4.1.311.21.10	875	16
1.3.6.1.4.1.311.21.7	873	312
privateKeyUsagePeriod	871	798
2.5.29.1	175	133
nsRevocationUrl	112	39
nsCaRevocationUrl	104	52
nsCaPolicyUrl	74	32
nsSslServerName	73	17
nsBaseUrl	63	31
1.2.840.113533.7.65.0	59	6
2.16.840.1.113719.1.9.4.1	54	26
nsRenewalUrl	33	7
2.5.29.80	10	10
qcStatements	8	2
2.5.29.7	7	7
2.16.840.1.113733.1.6.15	6	6
2.5.29.10	5	1
1.3.6.1.4.1.3401.8.1.1	4	4
freshestCRL	4	3
subjectDirectoryAttributes	4	2
1.3.6.1.4.1.311.10.11.11	3	3
2.5.29.3	2	1
2.16.840.1.113733.1.6.7	2	2
1.3.6.1.4.4324.33	2	2

Name or OID	Occurrences	Unique values
1.3.6.1.4.4324.36	2	2
1.3.6.1.4.4324.34	2	2
1.3.6.1.4.4324.35	2	1
1.2.40.0.10.1.1.1	2	2
1.3.6.1.4.1.311.21.1	2	1
1.3.6.1.4.1.7650.1	1	1
1.3.6.1.4.1.311.10.11.87	1	1
1.3.6.1.4.1.311.10.11.26	1	1
1.3.6.1.4.1.8173.2.3.6	1	1
1.2.40.0.10.1.1.2	1	1
2.5.29.4	1	1
1.2.250.1.71.1.2.5	1	1
1.3.6.1.4.1.6334.2.2	1	1

TABLE V

Semantic discrepancies in certificate validation (incorrect answers in **bold**)

Problem	Certificates triggering the problem occur in the original corpus	OpenSSL	PolarSSL	GnuTLS	CyaSSL	MatrixSSL	NSS	OpenJDK, Bouncy Castle	Browsers
Untrusted version 1 intermediate CA certificate	No	reject	reject	**accept**	reject	**accept**	reject	reject	reject
Untrusted version 2 intermediate CA certificate	No	reject	reject	reject	reject	**accept**	reject	reject	reject
Version 1 certificate with valid basic constraints	No	accept	reject	accept	accept	accept	reject	reject	Firefox: reject Opera, Chrome: accept
Intermediate CA not authorized to issue further intermediate CA certificates, but followed in the chain by an intermediate CA certificate	No	reject	reject	reject	reject	**accept**	reject	reject	reject
…followed by a leaf CA certificate	No	**reject**	**reject**	accept	**reject**	accept	**reject**	**reject**	**reject**
Intermediate CA not authorized to issue certificates for server's hostname	No	reject	reject	**accept**	**accept**	**accept**	reject	reject	reject
Certificate not yet valid	Yes	reject	**accept**	reject	reject	reject	reject	reject	reject
Certificate expired in its timezone	Yes	reject	**accept**	reject	reject	**accept**	reject	reject	reject
CA certificate not authorized for signing other certificates	No	reject	reject	**accept**	**accept**	**accept**	reject	reject	reject
Server certificate not authorized for use in SSL/TLS handshake	Yes	reject	**accept**	**accept**	**accept**	**accept**	reject	reject	reject
Server certificate not authorized for server authentication	Yes	reject	**accept**	**accept**	**accept**	**accept**	reject	reject	reject
Certificate with unknown critical extension	No	reject	reject	**accept**	**accept**	**accept**	reject	reject	reject
Certificate with malformed extension value	No	accept	reject	accept	accept	accept	reject	reject	reject
Certificate with the same issuer and subject and a valid chain of trust	No	reject	reject	**accept**	reject	**accept**	reject	reject	reject
Issuer name does not match AKI	No	reject	accept	accept	accept	accept	reject	reject	reject
Issuer serial number does not match AKI	No	reject	accept	reject	accept	accept	reject	reject	reject

TABLE VI

Error code(s) returned by Web browsers and SSL/TLS libraries for certificates with various combinations of Bad Issuer (I), Expired (E), and Bad Name (N). Security vulnerabilities are highlighted in **bold**.

Certs	Firefox 20	Chrome 30 (Linux)	Opera 12 (Linux)	Opera 20 (Mac)	Safari 7	Chrome 30 (Mac)	IE 10	OpenSSL	PolarSSL	GnuTLS	CyaSSL	MatrixSSL	NSS
E	E	E	E	!E	!E	E	E	E	E	E	E	E	E
I	I	I	I	!I	!I	I	I	I	I	I	I	**	I
IE	IE	**E**	I#	*	**!E**	*	*	I	I	IE	**	**	**E-**
IN	IN	I	I#	!I	!I	I	IN	I-	I-	I-	I-	*-	I-
IEN	IEN	**N**	I#	*	**!E**	*	*	I-	IE-	**-	**-	**-	**E-**
N	N	N	N	+	!N	N	N	-	-	-	-	-	-
NE	NE	N	**E#**	**!E**	**!E**	N	NE	E-	**-	E-	E-	E-	E-

* is a generic “invalid certificate” warning without a specific error message; the user cannot override this warning

+ is a generic “invalid certificate” warning without a specific error message; the user can override this warning

** is a generic “invalid certificate” error code

all errors are shown after the user clicks the details tab

! shows a generic error message first; the reported error is shown after user clicks the details button

- the hostname check was not enabled for any of the tested clients

TABLE VII

Support for cryptographic hash algorithms in certificate signatures

Algorithm	OpenSSL	PolarSSL	GnuTLS	CyaSSL	MatrixSSL	NSS	OpenJDK	BouncyCastle	Chrome	Firefox	WebKit	Opera
SHA-1	accept	accept	accept	accept	accept	accept	accept	accept	accept	accept	accept	accept
SHA-256	accept	accept	accept	accept	reject (u)	accept	accept	accept	accept	accept	accept	accept
SHA-512	accept	accept	accept	reject (u)	reject (u)	accept	accept	accept	accept	accept	accept	accept
MD2	reject	reject	reject	reject	reject	reject	reject	reject	reject	reject	reject	reject
MD4	reject	reject	reject	reject	reject	reject	reject (d)	reject	reject	reject	reject	reject
MD5	accept	accept	reject (w)	accept	accept	reject (w)	accept	accept	reject (w)	accept	accept	accept

reject (u) : reject because hash function is unknown

reject (w) : reject because hash function is weak

reject (d) : reject under default settings

TABLE VIII

Support for short keys and unusual key sizes

Key size	OpenSSL	PolarSSL	GnuTLS	CyaSSL	MatrixSSL	NSS	OpenJDK	BouncyCastle	Chrome	Firefox	WebKit	Opera
512-bit RSA	accept	accept	accept	accept	accept	accept	accept	accept	accept	accept	accept	warning
1023-bit RSA	accept	accept	accept	accept	accept	accept	accept	accept	accept	accept	accept	accept

TABLE IX

Verification of extra certificate fields

Library	CRL	subjectAltName	Host name
MatrixSSL	*	No	No
PolarSSL	Yes	Yes	Yes
CyaSSL	*	Yes	Yes
GnuTLS	Yes	Yes	Yes
NSS	Yes	Yes	Yes
OpenSSL	*	*	*

* not verified by default, application must explicitly enable