SymCerts: Practical Symbolic Execution For Exposing Noncompliance in X.509 Certificate Validation Implementations

Sze Yiu Chau* Omar Chowdhury† Endadul Hoque* Huangyi Ge* Aniket Kate* Cristina Nita-Rotaru‡ Ninghui Li*
{schau,mhoque,geh,aniket,ninghui}@purdue.edu, Purdue University, West Lafayette, Indiana, USA*
omar-chowdhury@uiowa.edu, The University of Iowa, Iowa City, Iowa, USA†
c.nitarotaru@neu.edu, Northeastern University, Boston, Massachusetts, USA‡

***Abstract*—The X.509 Public-Key Infrastructure has long been used in the SSL/TLS protocol to achieve authentication. A recent trend of Internet-of-Things (IoT) systems employing small footprint SSL/TLS libraries for secure communication has further propelled its prominence. The security guarantees provided by X.509 hinge on the assumption that the underlying implementation rigorously scrutinizes X.509 certificate chains, and accepts only the valid ones. Noncompliant implementations of X.509 can potentially lead to attacks and/or interoperability issues. In the literature, black-box fuzzing has been used to find flaws in X.509 validation implementations; fuzzing, however, *cannot* guarantee coverage and thus severe flaws may remain undetected. To thoroughly analyze X.509 implementations in small footprint SSL/TLS libraries, this paper takes the complementary approach of using *symbolic execution*.**

We observe that symbolic execution, a technique proven to be effective in finding software implementation flaws, can also be leveraged to expose noncompliance in X.509 implementations. Directly applying an off-the-shelf symbolic execution engine on SSL/TLS libraries is, however, not practical due to the problem of path explosion. To this end, we propose the use of *SymCerts*, which are X.509 certificate chains carefully constructed with a mixture of symbolic and concrete values. Utilizing SymCerts and some domain-specific optimizations, we symbolically execute the certificate chain validation code of each library and extract path constraints describing its accepting and rejecting certificate universes. These path constraints help us identify missing checks in different libraries. For exposing subtle but intricate noncompliance with X.509 standard, we cross-validate the constraints extracted from different libraries to find further implementation flaws. Our analysis of 9 small footprint X.509 implementations has uncovered 48 instances of noncompliance. Findings and suggestions provided by us have already been incorporated by developers into newer versions of their libraries.

I. Introduction

The X.509 Public-Key Infrastructure (PKI) standard [1], [2] has long been used in SSL/TLS as a means to distribute keys and provide authentication. The security assurance expected from SSL/TLS handshake critically hinges on the premise that communication peers, particularly the clients, correctly perform the prescribed validation of the server-provided X.509 certificate chain. Put differently, *correctly validating X.509 certificate chains is imperative to achieving security*. Flaws in implementations of the certificate chain validation logic (**CCVL**) could potentially lead to two pitfalls: (1) Overly restrictive CCVL (*i.e.*, incorrectly rejecting valid certificate chains) may result in **interoperability issues** and potential loss of service; (2) Overly permissive CCVL (*i.e.*, incorrectly accepting invalid certificate chains) may allow attackers to conduct **impersonation attacks**. We call an X.509 CCVL implementation **noncompliant with the X.509 specification** if it suffers from over-permissiveness, over-restrictiveness, or both. The X.509 standard [1] is defined in a generic way to accommodate different usage scenarios (*e.g.* for code signing, encipherment, authentication, etc.). In this work, we concentrate on X.509's use in the context of Internet communication (*i.e.*, clients performing server authentication during SSL/TLS negotiation) and focus on the RFC 5280 specification [2].

Although the SSL/TLS protocol implementations have undergone extensive scrutiny [3]–[8], similar rigorous investigation is absent for checking compliance of X.509 CCVL implementations. For instance, researchers have developed a formally verified reference implementation for the SSL/TLS protocol [7] but it does not include a formally verified CCVL. The portion of code in SSL/TLS libraries responsible for performing the X.509 chain validation are often plagued with severe bugs [9]–[21].

Implementing a compliant X.509 CCVL is not easy, primarily due to the complexity of its requirements. For example, through our analysis, we have seen how a supposedly simple boundary check on date and time can lead to various instances of noncompliance in different libraries due to mishandling time zones and misinterpreting the specification. The following comment from an SSL/TLS library developer that we contacted regarding a bug report concisely capture the intricacy of the task: "*In general, X.509 validation is one of the most error prone, code bloating, and compatibility nightmares in TLS implementation.*"

There are two possible directions for addressing X.509 CCVL's noncompliance problem: (1) Formally proving compliance of a (possibly reference) CCVL implementation with respect to the specification and having every library use it; (2) Devising approaches for finding noncompliance in CCVL implementations. The difficulty of automatically proving compliance of an X.509 CCVL implementation, in addition to the problem being undecidable in general [22], stems from

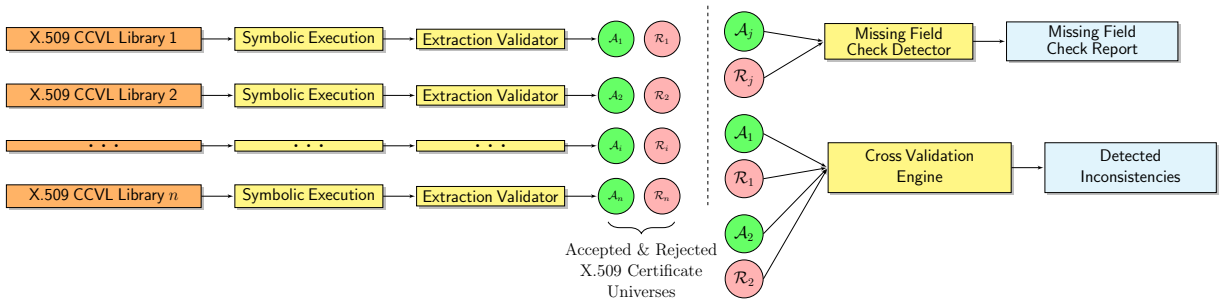

Fig. 1. Our noncompliance finding approach for X.509 CCVL implementations. **Symbolic execution engine** takes as input a CCVL implementation and extracts the approximated accepted and rejecting certificate universe whereas **extraction validator** validates it through concrete execution. **Missing field check detector** finds unscrutinized certificate fields from the universes. **Cross validation engine** performs cross validation among two implementations universes.

the fact that standard formal verification techniques [23]–[42] often do not support all the idiosyncrasies of a system level programming language like C. The direction of finding noncompliance was adopted by Brubaker *et al.* [43] and they uncovered a number of bugs in the CCVL implementations using black-box fuzzing, which raised awareness on both the existence and severity of the problem. Our approach is also geared towards finding noncompliance in real CCVL implementations.

Although black-box fuzzing is an effective technique for finding implementation flaws, especially when the source code is not available, it suffers from the following well known limitation: given a vast input space, black box fuzzing fails to concentrate on relevant portions of the source code without explicit guidance (*i.e.* lack of code coverage).

Symbolic execution [44] has been found to address the above limitation [45]–[47]. Symbolic execution is also known to be effective in finding bugs buried deep in the execution. It is, however, cursed by the problem of *path explosion* [48], which severely hinders its scalability and practicality, especially when the input is recursively structured and complex as in the case of X.509 certificates.

In this paper, we take the first step in making symbolic execution practical for finding noncompliance in real X.509 implementations. To this end, we solve symbolic execution's path explosion problem in the following manner: (1) Focusing our analysis on open source SSL/TLS libraries that have a small footprint and code base; (2) Applying a combination of domain-specific insights, abstractions, and compartmentalization techniques to the symbolic execution environment.

Small footprint SSL/TLS libraries are typically tailor-made for resource constrained platforms, and often prioritize efficiency over robustness. With the emergence of Internet-of-Things (IoT), these libraries are actively deployed on commodity devices to satisfy the needs for secure communication in the IoT ecosystem [49]–[52]. Furthermore, following the discovery of several high-profile vulnerabilities due to implementation flaws in recent years [53]–[55], traditional SSL/TLS libraries like OpenSSL has been criticized to have an unnecessarily large and messy code base that is both slow and infested with bugs [56]. A call for diverse alternative implementations with better maintainability and a desire for performance have sparked interests in adopting small footprint SSL/TLS libraries for building applications on even conventional PC platforms [57]–[61]. Hence it is of interest for us to evaluate these implementations of X.509 validation for robustness and compliance to specification.

To make symbolic execution practical and feasible, we develop the concept of **SymCerts**, which are syntactically well-formed symbolic X.509 certificate chains, such that each certificate contains a mix of concrete and symbolic values. To further reduce path explosion, we decompose the problem of noncompliance finding into smaller independent sub-problems based on the domain-specific observation that **some** certificate fields are logically independent in their semantic meanings. Fields in the same sub-problem are made symbolic at the same time, while the other unrelated fields are kept concrete. The use of SymCerts, along with the observation of semantic independence of fields, address the path explosion problem of symbolic execution that stem from the recursive and complex nature of the input certificate chain representation.

Approach: An X.509 CCVL *partitions* the certificate chain input universe into accepting (chains deemed valid) and rejecting (chains deemed invalid) certificate universes. We use symbolic execution to automatically extract the approximation of the certificate accepting and rejecting universes (See Figure 1), and symbolically represent these sets as path constraints (quantifier-free first order logic formulas), where the symbolic variables correspond to fields and extensions of certificates.

In the case where an X.509 CCVL implementation is noncompliant due to the lack of certain checks, a simple search (*e.g.* with `grep`) of the path constraints will uncover such noncompliance, as the corresponding symbolic variables will not appear in the extracted path constraints. For catching deeper noncompliance, we leverage the principal of *differential testing* [62], [63], by carrying out a cross validation of different implementations. Given two implementations I_1, I_2, and their corresponding accepting and rejecting certificate universes $\mathcal{A}_1$, $\mathcal{R}_1$, $\mathcal{A}_2$, and $\mathcal{R}_2$, we can automatically determine whether discrepancies exist between I_1 and I_2 (*i.e.*, one implementation accepts a certificate chain whereas the other rejects it) by checking whether the sets $\mathcal{A}_1 \cap \mathcal{R}_2$ and $\mathcal{A}_2 \cap \mathcal{R}_1$ are nonempty. Representing these sets symbolically enables us to implement the set intersection operator by leveraging a Satisfiability

Modulo Theory (SMT) solver [64], [65].

Evaluation and Findings: We analyzed 9 implementations from 4 families of code base (axTLS, wolfSSL, mbedTLS, MatrixSSL) and uncovered 48 instances of noncompliance.

Notably, we have detected the erroneous logic embraced by wolfSSL 3.6.6 and MatrixSSL 3.7.2 for matching ExtKeyUsage object identifiers (OID); such OID matching is used to assert the proper use of the key according to its intended purposes (*e.g.*, for code signing). Although standard usage purposes are identified with pre-defined values (*e.g.*, 1.3.6.1.5.5.7.3.1 means server authentication), other values are allowed for defining custom purposes. Both wolfSSL 3.6.6 and MatrixSSL 3.7.2 take a summation of the encoded bytes of an OID, and uses only the sum for matching against known standard key usage purposes. In their scheme, OID 1.3.6.1.5.5.7.3.1 (ASN.1 DER-encoded bytes: 0x2B 0x06 0x01 0x05 0x05 0x07 0x03 0x01) will be identified as decimal 71. Despite OIDs being unique hierarchically, the summation of their encoded bytes may not be. An adversary may request a certificate authority to issue an innocuous-looking certificate with a custom key usage purpose OID value that adds up to 71, and would then be able to use it for server authentication in these libraries. We have reported this bug to the library developers. They **acknowledged the problem and have it fixed** in new releases.

Another notable finding is the misinterpretation of the year field of UTCTime by MatrixSSL 3.7.2. In UTCTime format, the RFC prescribes two bytes YY to denote years such that YY$\in [0, 49]$ is treated as the year 20YY whereas YY$\in [50, 99]$ is treated as 19YY, allowing years to be in range $1950 - 2049$. However, MatrixSSL 3.7.2 misinterprets the YY field and hence miscalculates some certificate **expiration by** 100 **years** (*e.g.*, certificates expired in 1995 are considered to expire in 2095). Developers of MatrixSSL **acknowledged this bug** after receiving our report and **implemented a fix** in a newer version. Other findings are reported in Section VI.

Contributions: In summary, this paper makes the following contributions:

1) We take the first step towards developing a more principled approach to systematically analyze real implementations of X.509 validation.
2) Though scalability issue exists, we show that symbolic execution could be made practical by limiting the scope of analysis and using domain specific optimization, and it is very effective in exposing implementation flaws.
3) We revisit three specific implementations that have been studied before in the literature [43]. With new findings that are otherwise difficult to find with an unguided fuzzing approach, we show that previous work based on fuzz testing indeed suffers from false negatives, and some of their claims are inaccurate due to a possible misinterpretation of those false negatives.
4) For the other and more recent implementations that had not been studied before, we found multiple instances of noncompliance and have them reported to the developers.

II. Related Work

Given their prominence and importance, the research community has put implementations of the SSL/TLS protocols under close scrutiny in recent years. Here we give a brief overview of previous research efforts and account for how our work is different from them.

A. Forged certificates, attacks, and patching

Huang et al. [66] designed a client-side applet to monitor and report the certificates that were actually presented to clients. Their study discovered about 6 thousand forged certificates in over 3 million connections, and showed that not just malware but surveillance devices as well as anti-virus software are also forging certificates to tamper with SSL/TLS connections. A new attack on TLS known as the KCI attack has been found in [67]. This attack is possible due to the use of certain weak non-ephemeral cipher suites, plus the fact that installing end-entity (in contrast to CA) certificates do not trigger any warnings, and many implementations are not correctly handling the key usage extensions. *This also highlights the importance of correctly handling extensions when verifying a chain of certificates.* Bates et al. [68] proposed to use dynamically linked objects and binary instrumentation to implement a defense layer, so that vulnerabilities can be patched in a prompt manner, proper extension handling can be enforced, and insecure options can be overridden.

B. Incorrect and insecure usage of TLS library APIs

Georgiev et al. [69] crafted a handful of attack certificates to attempt MITM attacks against various SSL/TLS library-using applications, and showed that application developers often misunderstand and misuse APIs, resulting in vulnerabilities. Further discussions on false beliefs of developers, exploits on TLS-using applications and correct usage of TLS can be found in [70]. He et al. [71] showed how to use static analysis to vet and identify vulnerable API usage in applications. Yun et al. [72] propose a fully automated system called APISAN that can infer correct API usage from other usages to that API, and use this information to find inconsistent API usages.

The major difference between this line of research and our work is the different scope of focus. These research efforts focus on how application developers are making mistakes in terms of API calls to the libraries, while our work is focused on how the underlying SSL/TLS libraries providing those APIs are implementing the certificate validation logic. Problems in the libraries would affect applications even if application developers made no mistakes in using the APIs.

C. Fuzz testing of TLS implementations

Fuzzing has been a prominent approach in testing SSL/TLS implementations, where test cases are typically synthesized by applying mutation heuristics on known valid input (*e.g.* message sequences and certificates). Beurdouche et al. [3] looked at the problem of libraries mishandling unexpected sequences of messages when implementing support for various ciphersuites, authentication modes and protocol extensions.

Brubaker et al. [43] used black-box fuzzing to test client-side validation of X.509 certificates in SSL/TLS implementations. De Ruiter et al. [4] showed that the implemented state machine of SSL/TLS can be inferred by applying a fuzzing-based technique, which can then be verified manually to discover errors. A recent work by Somorovsky [5] presents a framework that allows developers to evaluate the behavior of TLS servers in a flexible manner, with the ability to create arbitrary protocol flows and dynamically modified messages.

Unguided black-box fuzzing, as used in previous work [43], makes a good first attempt to reveal the existence of problems in X.509 implementations of SSL/TLS libraries, especially when the source code is not available. However, there are limitations of such approach: 1) given a particular test case that indicates an error, it is not easy to account for the root causes; 2) it yields no guarantees on coverage of the code being tested; 3) each generated test case could contain multiple problems that might mask each other, making results difficult to interpret. Our work attempts to take advantage of the fact that when the underlying source code is available, one can infer useful information out of the code, and perform testing with better code coverage.

D. TLS state machine and high-confidence implementations

Attempts were made on building high-confidence TLS implementations with a focus on correct state transitions and cryptographic primitives, using re-engineered protocol specification and modular code base [6], as well as verified code along with security proofs [7]. Beurdouche et al. [8] designed a tool that uses a verified implementation as a reference to test the state machine of other SSL/TLS implementations.

State machine bugs are about how state transitions are being performed in response to a sequence of messages of the sub-protocols, whereas we investigate how the validation logic of X.509 certificates are being done, which is one crucial step in typical SSL/TLS handshakes, and have a broader scope of implications outside the context of SSL/TLS.

At the time of writing, previous work on reference SSL/TLS implementations do not include a formally verified X.509 CCVL. Possible future efforts made along this direction on building a high-confidence implementation of X.509 validation can be used as references to put verdicts on which behaviors are incorrect and noncompliant, given the discrepancies in libraries found by previous work [43] and this work.

E. Symbolic Execution

Symbolic execution has been shown to be effective for detecting low-level (memory) errors (*e.g.*, null dereferencing) [45]–[47], [73]–[79]. It has also been used for checking the equivalence of C functions [80], [81], for checking server–client interoperability of network protocols based on the set of packets accepted by them [82], for checking controllers in software-defined networks [83], [84], and for cross-checking different file system implementations to find semantic bugs [85]. The input arguments/messages considered in these work are structurally simpler than an X.509 certificate.

F. Symbolic Finite Automata

Symbolic finite automata (SFA) [86] extend finite automata by supporting symbolic inputs, that is, typical finite automata require the alphabet to be of finite size whereas a symbolic finite automata can support an infinite alphabet set. Recently, Argyros et al. [87], [88] proposed a black-box automata learning algorithm (*i.e.*, querying the program in a black box fashion) for extracting the SFA of a given program (*i.e.*, regular expression filters and string sanitizers), and use it to perform differential testing and find bugs in various implementations (*e.g.*, TCP, Web Application Firewalls). Our collected path constraints can be viewed as an unwinding of their learned SFA. The SFA learning algorithm requires sample transitions for each state to be **explicitly** given as an input. It is, however, not obvious how one would obtain such sample transitions for an X.509 implementation so that it can be given as input to the learning algorithm. Also, due to the white box nature of our analysis, our approach is likely to yield a more precise characterization of the implementation's internal logic.

III. Background and Problem Definition

In this section, we first present a brief introduction on X.509 certificates and their validation logic. We then present the noncompliance finding problem and the associated high level challenges.

A. Preliminary on X.509 Certificate Validation

The X.509 PKI standard is described in ITU-T Recommendation X.509 [1]. The certificate format itself, at the time of writing, has 3 versions. Version 2 and 3 were introduced to add support for certificate revocation lists (CRLs) and certificate extensions, respectively. X.509 certificates can be used in various environments for different purposes. A variety of standard certificate extensions are defined in the standard documents [1] and ANSI X9.55. RFC 5280 [2] profiles how version 3 certificates, extensions and CRLs are meant to be used specifically for the Internet. Since we focus on this particular prominent use case of X.509, in the rest of this section, we provide a simplified overview of what makes a certificate and how validation should happen in general, taking the viewpoint of an Internet client and using RFC 5280 as the main reference.

1) Contents of an X.509 certificate: At a very high level, a X.509 certificate is made of 3 parts: the `TBS` (To-Be-Signed) part, which includes most of the semantic content of the certificate; a signature algorithm identifier, which denotes the algorithm the issuer used to sign the certificate; and finally the actual signature value. The `TBS` part generally includes the following fields: version (version number), serialNumber (that can uniquely identify a certificate), signature (the signature algorithm identifier), issuer (name of the entity who signed the certificate), validity (a time period of which the certificate can be considered as valid), subject (name of the subject of the certificate), subjectPublicKeyInfo (the public key of the subject of the certificate).

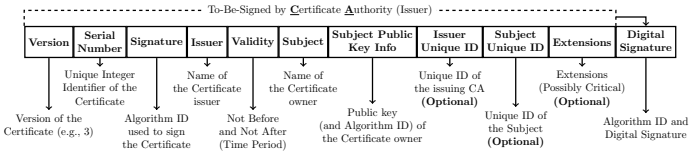

Fig. 2. A simplified structural view of an X.509 version 3 certificate (inspired by a similar figure in [89]).

Towards the end of the TBS of a X.509 version 3 certificate there are three optional constructs: the issuerUniqueID and subjectUniqueID, which are respectively unique identifiers of the issuer and subject of the certificate, followed by extensions, which is a sequence of X.509 version 3 extensions. See Figure 2 for a simplified visualization of the structure of a typical X.509 version 3 certificate.

2) X.509 certificate validation: The X.509 PKI is based on the idea of "chain of trust". The main objective of certificate validation is to show that given a trust anchor, C_0, the trust can be extended through a chain of certificates, all the way down to the communication peer (*e.g.* a specific server). Hence the basic check requires that for each certificate of a chain, the issuer name of a certificate C_i must equal to the subject name of the previous certificate C_{i-1}, and the signature on C_i can be correctly verified using the algorithm, the public key and other parameters derived from C_{i-1}.

In addition, each certificate involved in forming the chain of trust must be currently valid, in the sense that the current system time should be within the range (inclusively) prescribed by the notBefore and notAfter attributes of the Validity field.

Other checks in X.509 certificate validation are related to the handling of version 3 extensions. Extensions give CAs a means to impose additional restrictions on certificates issued by them, to avoid abuse of certificates.

Extensions can be marked as critical or non-critical. For the standard set of extensions, RFC 5280 [2] mandates some default criticality that a conforming CAs should follow. However, from the point of view of a certificate-using system, extensions should be processed regardless of their criticality if the system is able to, and in case it cannot process any of the critical extensions then the certificate should be rejected.

On a valid certificate chain, each of the certificates needs to be a CA certificate, except for the leaf one (both CA and non-CA are allowed). In X.509 version 3, this is achieved by checking the basicConstraints extension, which contains an isCA boolean field indicating whether the certificate is a CA certificate or not, and an optional integer pathLenConstraint that limits the number of non-self-issued intermediate CA certificates that can follow on the chain, not counting the leaf one. Before version 3, X.509 certificates do not have extensions. In such cases, clients can choose to either consider those to be non-CA certificates, or use an out-of-band mechanism to verify if those are CA certificates or not.

The KeyUsage and ExtKeyUsage are two useful extensions that describe the intended purposes of a certificate. With issuing CAs imposing these on certificates, and clients faithfully checking the intended purposes, some certificate abuse scenarios can be stopped (*e.g.* using a certificate that is only issued for signing software in a SSL/TLS handshake for authentication would not be allowed).

There are other standard extensions which we do not present here. For a complete list of extensions deemed useful for the Internet, and the details on how to handle them, we refer the readers to RFC 5280 [2].

3) Sources of noncompliance: The intricacies of implementing a compliant X.509 CCVL stem from the rich set of fields in certificates, which are further complicated by their wide range of possible values, as well as the numerous optional but possibly critical extensions. Noncompliance can occur due to the following two reasons:

a) Certain fields and/or extensions that must be checked are not involved in the decision making procedure of a CCVL implementation. This can be further divided into:
   i) The fields and/or extensions are not being parsed into an internal data structure. This is mostly due to a lack of intention to support a thorough and robust check, possibly due to concerns on resource usage.
   ii) The fields and/or extensions are being parsed into an internal data structure but checks did not happen. This is mostly due to an intention to perform the checks but the implementation is not complete.

b) The fields and/or extensions are involved into deciding whether to accept or reject the chain, but due to coding and/or logical errors in the parsing code and/or validation code, the checks are not performed correctly.

B. Goal and Challenges

In this paper, our goal is to **check whether a given X.509 CCVL implementation is compliant with the X.509 specification**. There are two ways to go about checking compliance of an implementation, namely, (1) proving the compliance of the implementation with respect to the specification and (2) trying to find noncompliance in the implementation. Our approach is geared towards finding noncompliance.

*1) **Why Not Prove Compliance**:* To prove compliance of a given CCVL implementation, we have to formally specify the valid sets of X.509 certificate chains that a CCVL implementation should accept. The X.509 specification is, however, described in natural languages and coming up with a complete formal specification is cumbersome and error-prone. Furthermore, even if we have such a formal specification Ψ at our disposal, proving that Ψ is satisfied by the CCVL implementation I (*i.e.*, $I \models \Psi$) using standard formal verification techniques [23]–[42] is infeasible as the problem is undecidable in general [22]. Also, formal verification techniques often do not scale and support real implementations. For this reason, we resort to noncompliance finding in the implementation.

*2) **Challenges**:* We now discuss the inherent challenges of the noncompliance finding.

Natural Languages Specification: The X.509 specification is written in English and it is inherently prone to underspecification, ambiguities, inconsistencies, and misinterpretations. To validate a noncompliant instance it is often required

to consult the specification when we do not have a formal specification at our disposal. We resort to manual effort to address this challenge.

Scalability: The complex format of X.509 certificates and also the intricacies in certificate chain validation make it difficult to develop a scalable noncompliance checker. Also, it is difficult to develop a scalable noncompliance checker for real libraries written in system level languages such as C.

Cryptographic Libraries: A X.509 CCVL relies on cryptographic functions to perform operations such as digital signature verification. Cryptographic functions are well recognized to be difficult to automatically analyze for correctness.

IV. Our Noncompliance Finding Approach

In this section, we first briefly describe symbolic execution and then present how we leverage it for noncompliance detection. Finally, we discuss several technical challenges of applying symbolic execution and how we overcome them.

A. Preliminary on Symbolic Execution

Symbolic execution [44] has been shown to be an effective way of inferring test cases that yield high code coverage [45]–[47], [73]–[79]. It achieves this objective by running a program with symbolic values for input variables. During execution, when it encounters a branch instruction (*e.g.*, if-else) with a branching condition on symbolic values, it consults a Satisfiability Modulo Theory (SMT) solver [64], [65] to check whether any of the two branches (*i.e.*, the if and else branches) are possible according to their branching conditions. If any of the branches are feasible (*i.e.*, the branching conditions are satisfiable for some concrete values for the input variables), the execution explores the corresponding paths. It keeps collecting all the feasible branching conditions on the input (symbolic) variables, also known as *path constraints*, until the program terminates or reaches a point of interest (*e.g.*, an error location). It then consults an SMT solver to obtain concrete values for the input that will induce the path in question.

B. Approximating Universes with Symbolic Execution

For noncompliance detection, our approach critically relies on extracting the universes of accepted and rejected certificate chains induced by a given X.509 CCVL implementation.

Suppose we denote the universe of all possible X.509 certificate chains with $\mathcal{C}$, a given X.509 CCVL partitions $\mathcal{C}$ into two sets $\mathcal{A}$ (the set of accepting certificate chains) and $\mathcal{R}$ (the set of rejecting certificate chains) such that $\mathcal{C} = \mathcal{A} \cup \mathcal{R}$ and $\mathcal{A} \cap \mathcal{R} = \emptyset$. To detect noncompliance in a given X.509 CCVL implementation, we automatically extract the sets $\mathcal{A}$ and $\mathcal{R}$. Due to the large number of possible certificate chains, explicitly enumerating elements of the sets $\mathcal{A}$ and $\mathcal{R}$ is not feasible. We represent the sets $\mathcal{A}$ and $\mathcal{R}$ symbolically by a set of quantifier-free first order logic (QFFOL) formulas [64] $\{f_1, f_2, \ldots, f_n\}$ where each QFFOL formula f_i represents a set of concrete certificate chains. We choose QFFOL as it is sufficiently expressive and also decidable for certain theories (*e.g.*, bitvector, array)—one can leverage an SMT solver to detect noncompliance—whereas the full first order logic (FOL) is undecidable. We use the theory of bitvectors and array.

For a given X.509 implementation, we extract the sets $\mathcal{A}$ and $\mathcal{R}$ by symbolically executing the CCVL of that given implementation with respect to a symbolic certificate chain. Symbolically executing the CCVL can capture the validation logic for that given implementation through path constraints and their associated return values of the CCVL function. The path constraint in question here contains input variables coming from the input certificate chain that has fields and extensions we marked to have symbolic values. Given a collected path constraint f and its associated boolean value b returned by the CCVL function, if $b = \mathsf{true}$ (resp., false), it signifies that any concrete certificate chain c that satisfies the constraint f (*i.e.*, $c \models f$) is accepted (resp., rejected) by the given CCVL. Precisely, after symbolic execution of the CCVL, we have $\mathcal{C} = \{\langle f_1, b_1\rangle, \langle f_2, b_2\rangle, \ldots, \langle f_n, b_n\rangle\}$ where f_i is a path constraint (*i.e.*, QFFOL formula) we obtained during symbolic execution of the CCVL and $b_i \in \{\mathsf{true}, \mathsf{false}\}$ is the return value of the CCVL function for the path constraint f_i. From $\mathcal{C}$, we construct $\mathcal{A}$ and $\mathcal{R}$ in the following way: $\mathcal{A} = \{f_i \mid \langle f_i, \mathsf{true}\rangle \in \mathcal{C}\}$ and $\mathcal{R} = \{f_j \mid \langle f_j, \mathsf{false}\rangle \in \mathcal{C}\}$.

The sets $\mathcal{A}$ and $\mathcal{R}$ induced by a given X.509 CCVL implementation are **the core asset of our noncompliance detection approach**. Given the sets of $\mathcal{A}_{\mathsf{test}}$ and $\mathcal{R}_{\mathsf{test}}$ induced by a CCVL implementation under test I_{test} and the sets $\mathcal{A}_{\mathsf{standard}}$ and $\mathcal{R}_{\mathsf{standard}}$ induced by the X.509 standard specification (*e.g.*, RFC), I_{test} is noncompliant if one of the following (or, both) hold: (1) $\mathcal{A}_{\mathsf{test}} \neq \mathcal{A}_{\mathsf{standard}}$ and (2) $\mathcal{R}_{\mathsf{test}} \neq \mathcal{R}_{\mathsf{standard}}$. For a given I_{test}, we can use its $\mathcal{A}_{\mathsf{test}}$ and $\mathcal{R}_{\mathsf{test}}$ to expose noncompliance in several ways, possibly by leveraging an SMT solver [65], [90]. We discuss them presently.

C. Approaches for Exposing Noncompliance

We now discuss three approaches where we leverage symbolic execution and the sets $\mathcal{A}$ and $\mathcal{R}$ to find noncompliance in X.509 CCVL implementations.

1) Noncompliance during Symbolic Execution: During symbolic execution of the X.509 CCVL function of a given implementation, the symbolic execution engine can discover certain low level memory errors (*e.g.*, array out of bounds). We have discovered an erroneous extension processing bug using this approach. We present the details in Section VI.

2) Simple Searching of the Path Constraints: By inspecting all the path constraints in the set $\mathcal{A} \cup \mathcal{R}$ for a particular CCVL implementation, one can easily notice missing checks of certain certificate fields. Let us assume that we assigned the subject name field of a certificate to have the symbolic value sym_sub_name. We can then perform a search with the string sym_sub_name (*i.e.*, often a simple `grep` will suffice) among all the path constraints in $\mathcal{A} \cup \mathcal{R}$. If the search turns up empty, one can conclude with high confidence that the implementation does not check the subject name field. This approach enables exposure of noncompliance due to an implementation's inability to take certificate fields into

consideration during the CCVL decision making process. We have discovered several serious noncompliances using `grep`.

3) Cross Validation: To expose deeper noncompliant instances—the ones due to an implementation's inability to impose proper validity checks on a certificate field even after recognizing it—ideally we want the sets $\mathcal{A}_{\mathsf{standard}}$ and $\mathcal{R}_{\mathsf{standard}}$ induced by the X.509 standard specification. We have, however, neither a formally verified CCVL implementation we can extract the sets $\mathcal{A}_{\mathsf{standard}}$ and $\mathcal{R}_{\mathsf{standard}}$ from, nor a formal specification for X.509 CCVL at our disposal. We compensate for the lack of the sets $\mathcal{A}_{\mathsf{standard}}$ and $\mathcal{R}_{\mathsf{standard}}$ by utilizing the existence of a large number of open source SSL/TLS library implementations. We can perform a cross validation (or, differential testing [62], [63]) by pitting the different implementations against each other. If two implementations come to different conclusions about whether a given certificate chain is valid, even though it is not clear which implementation is noncompliant, we can conclude that one of the libraries is noncompliant. Precisely, for any two implementations I_1 and I_2 and their corresponding sets $\mathcal{A}_1$, $\mathcal{R}_1$, $\mathcal{A}_2$, and $\mathcal{R}_2$, any $c \in \mathcal{C}$ such that (1) $c \in \mathcal{A}_1 \wedge c \in \mathcal{R}_2$ or (2) $c \in \mathcal{A}_2 \wedge c \in \mathcal{R}_1$ represents an instance of noncompliance.

One can utilize the path constraints from two different implementations to find inconsistent conclusions in the following two ways. In our analysis, we follow approach 2.

Let us assume for any two given implementations I_p and I_q, we have the following sets:

$$\mathcal{A}_p = \{a_1^p, a_2^p, \ldots, a_n^p\} \quad \text{(accepting certificate universe of } I_p\text{)}$$
$$\mathcal{R}_p = \{r_1^p, r_2^p, \ldots, r_m^p\} \quad \text{(rejecting certificate universe of } I_p\text{)}$$
$$\mathcal{A}_q = \{a_1^q, a_2^q, \ldots, a_s^q\} \quad \text{(accepting certificate universe of } I_q\text{)}$$
$$\mathcal{R}_q = \{r_1^q, r_2^q, \ldots, r_t^q\} \quad \text{(rejecting certificate universe of } I_q\text{)}$$

Approach 1: To detect inconsistencies between I_p and I_q, one can check to see whether either of the following formulas is satisfiable: $\neg(\bigvee_{(1\leq i\leq n)} a_i^p \leftrightarrow \bigvee_{(1\leq j\leq s)} a_j^q)$ and $\neg(\bigvee_{(1\leq i\leq m)} r_i^p \leftrightarrow \bigvee_{(1\leq j\leq t)} r_j^q)$ ($\leftrightarrow$ stands for logical equivalence). The first (resp., second) formula asserts that the accepting (resp., rejecting) paths of I_p and I_q are not equivalent. Any model of either of the formulas will signify a noncompliant instance. We, however, do not utilize this approach to detect noncompliance for the following three reasons: (1) For each satisfiability query the SMT solver will present one model (*i.e.*, one noncompliant instance) even in the presence of multiple noncompliant instances (**We desire as many noncompliant instances instead of just one at a time**); (2) The resulting formulas are large and it may put heavy burden on the SMT solver; (3) Due to the incompleteness caused by techniques used to relieve path explosion, the extracted sets $\mathcal{A}$ and $\mathcal{R}$ may not be exhaustive (*i.e.*, complete), yielding false positives.

Approach 2: In this approach, we first take each accepting path a_i^p from $\mathcal{A}_p$ and each rejecting path r_j^q from $\mathcal{R}_q$ where $1 \leq i \leq n, 1 \leq j \leq t$, and check to see whether the formula $a_i^p \wedge r_j^q$ is satisfiable by consulting an SMT solver. If the formula is satisfiable, it signifies that there is at least one certificate chain that I_p accepts but I_q rejects. The model obtained for the formula from the SMT solver, can be used to construct a concrete certificate chain signifying an evidence of inconsistency. We can then repeat the same process by taking each accepting path from I_q and each rejecting path from I_p. Note that, multiple pairs may induce inconsistencies due to the same noncompliant behavior and sometimes best-effort manual analysis of the source code is needed to detect the root cause.

D. Scalability Challenges of Applying Symbolic Execution

The application of symbolic execution in a straightforward way to extract the sets $\mathcal{A}$ and $\mathcal{R}$, considering all certificates in the chains and other arguments to the CCVL function to have symbolic values, will not yield a scalable noncompliance detection approach. *Our feasibility evaluation have verified this observation*. We have also tried only one of the certificates in the chain to have symbolic values and even then the symbolic execution did not finish due to resource exhaustion. The scalability problem is predominantly due to *symbolic value dependent loops*—loops whose terminating conditions depend on symbolic values—in the certificate parsing implementation. One way to get around this challenge is to assume the correctness of the parsing code and just focus on the core CCVL logic. Ignoring the parsing logic, however, is not sufficient to capture the majority of the CCVL logic as some of the sanity checks on the certificate fields are done during parsing. In addition, capturing only the CCVL logic would require one to manually modify the internal data structure where the certificate fields are stored after parsing. This approach requires significant manual efforts (*i.e.*, code comprehension) and is also error-prone.

E. Our Solution—SymCerts and Problem Decomposition

For addressing the scalability challenge we rely on carefully crafting symbolic certificates and also on our domain specific observations. Rather than extracting the complete sets $\mathcal{A}$ and $\mathcal{R}$, we use domain-specific observations and specially crafted symbolic certificate chains to extract an approximation of the sets $\mathcal{A}$ and $\mathcal{R}$, *i.e.*, $\mathcal{A}_{\mathsf{approx}}$ and $\mathcal{R}_{\mathsf{approx}}$. Our approximation has both under- and over-approximation. To overcome path explosion, we create a chain of **SymCerts** where some portions of each certificate have concrete values whereas the others have symbolic values. SymCerts along with the following observation aid in achieving scalability during the extractions of the sets $\mathcal{A}_{\mathsf{approx}}$ and $\mathcal{R}_{\mathsf{approx}}$ from an X.509 CCVL.

One domain specific observation we use is the **logical independence between certificate fields**. For instance, the logic of checking whether a certificate is expired according to its notAfter field is independent of the logic of checking whether a certificate's issuer name matches with the subject name of the predecessor certificate in the chain. In this case, we can try to capture the logic of checking certificate expiration independently of the checking of issuer and subject names. Based on the notion of independence, we group the certificate fields into equivalence classes where the logic of fields in the same equivalence class should be extracted at the

same time, that is, fields in the same equivalence class should be marked to have symbolic values at the same time. We leverage this observation by generating a SymCert chain for each equivalence class where each element of the equivalence class has symbolic values whereas the rest of the fields have concrete values. **Note that we certainly do not claim that the checking logic of all certificate fields are independent**; there are obviously certificate fields whose value influences one another. For instance, the value of the isCA field of an X.509 certificate (*i.e.*, whether the certificate is a CA certificate) prescribes certain corresponding key usage purposes (*i.e.*, affecting the KeyUsage extension). In this case, the isCA field needs to be in the same equivalence class as KeyUsage.

In our analysis, we conservatively partition the certificate fields into 2 equivalence classes. We refer to these two equivalence classes as EqC_1 and EqC_2, respectively. EqC_1 has all the relevant certificate fields symbolic, except the Validity date time period fields which are symbolic only in EqC_2.

V. Implementation

In this section, we discuss additional challenges of applying symbolic execution to CCVL code, and our approach to addressing these challenges. We also discuss other aspects of implementing our noncompliance finding approach.

Challenge 1 *(Complex Structure of X.509 Certificates):* X.509 certificates are represented in the *Abstract Syntax Notation One* (ASN.1) [91], [92] notation. X.509 certificates are typically transmitted in byte streams encoded following the *DER* (Distinguished Encoding Rules), which are binary in nature. Under the DER format, an X.509 certificate has the form $\langle t, \ell, v\rangle$ where t denotes a type, ℓ denotes the length of the values in bytes, and finally v represents the value. t can represent complex types such as a sequence where the value v can be recursively made of other $\langle t, \ell, v\rangle$ triplets. Such nesting of $\langle t, \ell, v\rangle$ triplets inside a v field can be arbitrarily deep.

The problem of marking the whole certificate byte-stream as symbolic is that, during certificate parsing, the symbolic execution engine will try different values for ℓ as it is symbolic, and the parsing code will keep reading bytes without knowing when to stop. This will cause memory exhaustion.

Approach—SymCerts (Certificates With Symbolic and Concrete Values): To avoid the scalability problem, instead of using a fully symbolic certificate chain, we develop a certificate chain in which each certificate byte-stream contains some of concrete values and some symbolic values. We call each such certificate a **SymCert**.

We construct a SymCert in the following way: For each leaf $\langle t, \ell, v\rangle$ tuple (*i.e.*, v contains a value instead of another $\langle t, \ell, v\rangle$ tuple) in a certificate byte-stream, we ensure that the fields t and ℓ have concrete values whereas only the v field is symbolic. Concrete values of t can be obtained from actual certificates and we use them as the backbone for generating SymCerts. For the l field, we consult the RFC document to select appropriate concrete values. For instance, when marking the OIDs used in the ExtKeyUsage extension symbolic, we give it a concrete length of 8, as most of the standard key usage purposes defined in RFC 5280 [2] are 8-byte long.

Due to the complexity of DER byte-streams, it is difficult for a user to directly manipulate and construct SymCerts from scratch. In addition, due to nesting, changing the length field (*i.e.*, ℓ) of a child $\langle t, \ell, v\rangle$ triplet may require adjustment on the length field (*i.e.*, ℓ) of the parent $\langle t, \ell, v\rangle$ triplet. For this, we developed a Graphical User Interface (**GUI**), by extending the ASN.1 JavaScript decoder [93]. Our GUI allows a user to see and click on different certificate fields, so that they can be replaced with a desired number of symbolic bytes, and the new length will be correctly adjusted. The GUI will then automatically generate code that can be used for symbolic execution. We use OpenSSL to generate concrete certificate chains as the input to our GUI, which constitute the basis of our SymCerts. The philosophy here is that all major fields (*e.g.* optional extensions, criticality booleans) of a certificate need to be explicitly available on the base input certificate, as it is difficult to mark nonexistent fields symbolic.

Challenge 2 *(System Time Handling):* Given that our symbolic execution of the implementations would happen at different times, if we simply allow the implementations to use the local system time, then the constraints we have extracted would not be comparable, as the system time elapses.

Approach—Constant Static Time: We consider a fixed concrete time value for the system time. We use the same concrete value for these inputs during the analysis of all implementations. Using a symbolic variable is also possible, but using concrete values has the advantage of reducing the complexity of the path constraints which consequently improves scalability.

Challenge 3 *(Cryptographic Functions):* The cryptographic functions (*e.g.*, for verifying digital signatures) called by the CCVL contain loops dependent on symbolic data, which severely impact the scalability of symbolic execution.

Approach—Cryptographic Stub Functions: We abstract away the cryptographic functions with stub functions. For instance, the function that matches the digital signature of a certificate is abstracted away by a stub function that returns True indicating the match was successful. In this work, we consider cryptographic correctness beyond our scope. Instead, we are interested in finding out what fields are checked and what restrictions are imposed on these fields.

Challenge 4 *(Complex String Operations):* As part of the CCVL, implementations are sometimes required to perform complex string operations (*e.g.*, wild card matching, null checking) on certificate fields such as subject name and issuer name. Faithfully capturing the string operations with QF_BVA logic (*i.e.*, QFFOL formulas with equality, bit vector, and array theories)—which is the underlying logic of the symbolic execution engine we use—does not scale well.

Approach—Single Byte Strings: We consider names and other string-based certificate fields to have a single byte symbolic value, which significantly improves the scalability. However, because of this, our analysis misses out on finding noncompliance due to erroneous string operations.

Challenge 5 *(Hashing for Checking Multi-Field Equality):* When checking the equality of two name fields of certificates—name fields are compound fields containing the following sub-fields such as street address, city, state/province, locality, organizational name & unit, country, common name—some implementations take a hash of the concatenation of all the sub-fields and match the hash values, instead of checking the equality of each sub-field. Trying to solve the constraints from such a match would be similar to attacking the hash collision problem, which is not scalable to analyze with symbolic execution due to symbolic data-dependent loops.

Approach—Hash Stub: The hash function in question (*i.e.*, SHA-1) returns a 20-byte hash value. We replace it with a SHA-1 stub which returns a 20-byte value where the (symbolic) name sub-fields are packed together. Because of the single byte approach we introduced to simplify string operations described in the previous challenge, 20-byte is more than enough to pack all name sub-fields of interests.

Challenge 6 *(Certificate Chain Length):* While symbolically executing the CCVL of a given implementation, one natural question that arises is: "How many certificates in the symbolic certificate chain should we consider?" An X.509 CCVL implementation often parses the input X.509 certificate chain first and then checks the validity of different fields in the certificates of the parsed chain. During symbolic execution, if the execution detects a loop whose terminating condition relies on a symbolic value, it faces the dilemma of how many times to unroll the loop. Such loops in the implementation often cause path explosion in symbolic execution, resulting in incompleteness and scalability challenges. If we consider the certificate chain length to be symbolic, then the symbolic execution, especially during parsing, would try all possible values for the chain length, causing memory exhaustion.

Approach—Concrete Chain Length: For majority of our analysis, we consider a certificate chain of length 3 such that one of the certificates is the root CA certificate, the other is an intermediate CA certificate, and finally the remaining certificate is the certificate of the server currently being authenticated. While analyzing the logic of checking the path length constraint of the basic constraint extension, we also consider certificates with chain length 4 where we have two intermediate CA certificates.

Challenge 7 *(Other aspects of Path Explosion):* After the simplifications described above, the symbolic execution engine still generates a large number of paths. We especially observed that making all the v values of $\langle t, \ell, v\rangle$-tuples that represent certificate fields and extensions symbolic yields a lot of paths.

Approach—Early Rejection and Grouping Fields: We observed that implementations sometimes do not return early even in the case one of the certificates cannot be parsed or one of the fields validity checks failed. This contributes to a multiplicative factor to the number of paths. We judiciously applied early rejection when parsing or validation check fail. Finally, we applied the *logical independence between certificate fields* based on their semantics to decompose the noncompliance finding fields. We generated two equivalence classes, one consists of time fields related to the certificate Validity period checking, whereas the other contains all the remaining fields. One could possibly employ a more aggressive grouping of fields that need to be check together. We, however, make a conservative choice because if developer incorrectly introduces artificial dependencies in the implementation, we would like to capture them as well.

Challenge 8 *(Time Field Comparison):* An X.509 certificate contains two time fields (*i.e.*, notBefore and notAfter) which are compared to the current system time. A time field can be represented in two formats (*i.e.*, GeneralizedTime and UTCTime). In GeneralizedTime, the time field contains a 15-byte ASCII string where day, month, hour, minute, second contribute 2 bytes each; year contributes 4 bytes, and 1 byte is used to represent the time zone. For UTCTime, the only difference is that year contributes 2 bytes instead of 4. Sanity checks are often performed to ensure the fields are well-formed (*e.g.*, for minute, the most significant digit cannot be larger than 6). Marking the format symbolic and let the symbolic execution engine choose the length of the ASCII string contributes to poor scalability.

Approach—Decomposing Time Fields: In addition to checking noncompliance in time fields handling independently from other fields, we further decompose the analysis by analyzing the two time formats separately. We use two different SymCerts during symbolic execution, one with UTCTime and the other with GeneralizedTime, using the concrete length of the date time ASCII string according to the format.

Challenge 9 *(Redundant Pair of Paths in Cross-Validation):* When cross-validating two implementations I_p and I_q, the upper bound of discrepancies is $|\mathcal{A}_p| \times |\mathcal{R}_q| + |\mathcal{A}_q| \times |\mathcal{R}_p|$. Based on the number of paths in accepting (*e.g.*, $\mathcal{A}_p$ and $\mathcal{A}_q$) and rejecting (*e.g.*, $\mathcal{R}_p$ and $\mathcal{R}_q$) universes, the maximum number of noncompliance instances can be fairly large which creates a challenge for manual inspection to identify the root cause of the noncompliance.

Approach—Iterative Pruning: We observe that many pairwise discrepancies are due to the same root cause. Suppose implementation I_p does not check a particular field that I_q checks. In this case, the missing check in I_p's accepting path will likely be enumerated through many rejecting paths of I_q, resulting in a large number of redundant noncompliance instances. To make it easier to analyze the results of cross-validation, once we have identified such a case, we can concretize the value of that specific field, repeat the extraction step and continue cross-validation with a pruned search space.

Challenge 10 *(False Positives):* Due to different domain specific simplifications and the fact that we are abstracting away cryptographic functions, our approach can yield false positives, predominantly due to the path constraint extraction might not be capturing the real execution faithfully. In addition, the specification (*i.e.*, RFC document) states some fields should be checked by a certificate using system, without imposing whether the library or application (the two of them constitute the system) should perform each check. Consequently, SSL/TLS libraries have different API designs due to

such unclear separation of responsibility. Some libraries might enforce all the checks during certificate chain validation, while some might not and instead provide optional function calls for application developers desiring such checks, and the other libraries might completely delegate the task of implementing such checks to the application developers. As a clear boundary cannot be drawn easily, false positives can arise if some optional but provided checks are missed out during extraction.

Approach—Concrete Replay: To avoid false positives, we use a real client-server setup to help us verify our findings. We capitalize on the fact that a minimalistic sample client code is often made available in the source tree by library developers to demonstrate how the library should be used in application development and use such clients to draw the baseline. To gain confidence that our extracted path constraints adequately capture the real execution, for each accepted (resp., rejected) path constraint, we consult the SMT solver to obtain a concrete certificate chain and feed it to a real client-server setup to see whether the client would actually accept (resp., reject) the chain. This helps us to see whether the real execution concurs with our extraction. Similarly, during cross validation between implementations I_p and I_q, for the discrepancies we found (in the form of a model provided by the SMT solver), we construct a concrete certificate chain out of the model and use the client-server setup to verify it is indeed the case that I_p would accept and I_q would reject the chain.

VI. Evaluation and Results

We applied our approach in testing 9 open-source implementations from 4 major families of SSL/TLS library source trees, as shown in Table I. Implementations that have been tested in previous study by Brubaker et al. [43] are prefixed with an asterisk. These libraries have seen active deployments in embedded systems and IoT products to satisfy the security needs for connecting to the Internet (*e.g.* axTLS in Arduino [51] and MicroPython [52] for ESP8266, mbedTLS, tropicSSL and MatrixSSL on Particle hardware [49], [50], etc.), and are sometimes used even in building applications and libraries on conventional desktop platforms [57]–[61], due to their performance and small footprint advantage. We test multiple versions of a library from the same family in order to compare with previous work, and to see if the more recent versions implement a more complete and robust validation logic.

In this section we first show statistics that justify the practicality of our approach, and then present noncompliance findings grouped by how we uncovered them along the 3 approaches described in Section IV-C, together with other discrepancies and observations that we made while working with the libraries. Findings on recent versions of the implementations, whenever applicable, are reported to the corresponding developers. Many of our reports had led to fixes being implemented in newer versions.

A. Implementation Efforts and Practicality

For our analysis, we used the KLEE symbolic execution engine [45] and the STP SMT solver [65]. We added around 2000 lines of C++ code for implementing the path constraint extraction and cross validation engines, around 500 lines of Python for parsing path constraints and automating concrete test case generation, and around 400 lines of HTML plus less than 300 lines of JavaScript for the GUI that enables the easy construction of SymCerts.

In order to implement the various optimizations described before, a limited amount of new code need to be added to the libraries that we tested. As shown in Table I, no more than 75 lines of code were added to each of the library. Most of the new code is used to implement a static system time (see Section V-Challenge 2) and a stub cryptographic signature check (Section V-Challenge 3). Additionally, for CyaSSL 2.7.0, wolfSSL 3.6.6, and MatrixSSL 3.7.2, some code was added to implement the hash stub (see Section V-Challenge 5). PolarSSL 1.2.8 and tropicSSL needed a simplified version of `sscanf()`, and axTLS (both 1.4.3 and 1.5.3) needed a simplified version of `mktime()`, to avoid symbolic-data dependent loops, both of which are used for reading in and converting the format of date-time inputs.

Also shown in Table I are the performance statistics regarding path extraction. We ran our experiments on a commodity laptop equipped with an Intel i5-2520M CPU and 16GB RAM. Path extraction using EqC_1 for most implementations finished in minutes, while for some heavier ones it completed in hours. The total number of paths ranges from hundreds to the level of ten thousands. For EqC_2, we report the upper bound of the total number of paths, referred to in the table as "Total Paths", because the actual number could vary within each library due to different treatments (and possibly missing checks) for UTCTime and GeneralizedTime (see Section VI-C and VI-D for examples). For each library, extraction using EqC_2 yielded paths at the scale of tens, and finished within a minute.

B. Errors Discovered By Symbolic Execution

The first opportunity our approach provides is that, during symbolic execution, certain low-level coding issues (*e.g.* memory access errors, division by zeros, etc.) could be found.

Finding 1 *(Incorrect extension parsing in CyaSSL 2.7.0 [1]):* As shown in Listing 1, due to a missing `break` statement after `DecodeAltNames()`, the execution falls through to the next case and also invokes `DecodeAuthKeyId()`. Consequently, some bytes of the subject alternative name extension, which we made symbolic, will overwrite the authority key identifier (a pre-computed hash value) at the time of parsing. The error manifests later during certificate chain validation, when the authority key identifier undergoes some bit shifting operations and modulo arithmetic, effectively turning it into an array accessing index, which is then used to fetch a CA certificate from a table of trusted CA certificates. Since some bytes of the authority key identifier were incorrectly made symbolic during parsing, the execution engine caught potential memory access errors in fetching from the table. This was not reported in [43], which applied fuzzing to test CyaSSL 2.7.0. Our conjecture is

[1]This bug has been fixed in newer versions of CyaSSL and wolfSSL.

TABLE I
PRACTICALITY AND EFFICACY OF APPLYING THE SYMCERT APPROACH IN TESTING VARIOUS SMALL FOOTPRINT SSL/TLS LIBRARIES

Library - version	Released	Lines of C code in library	Lines Added	Paths $[\mathsf{EqC}_1]$	Extraction Time $[\mathsf{EqC}_1]$	Total Paths $[\mathsf{EqC}_2]$	Extraction Time $[\mathsf{EqC}_2]$	Found Instances of Noncompliance
axTLS - 1.4.3	Jul 2011	16,283	72	276 (419)	$\sim$ 1 Minute	≤ 52	$\leq$ 1 minute	7
axTLS - 1.5.3	Apr 2015	16,832	69	276 (419)	$\sim$ 1 Minute	≤ 52	$\leq$ 1 minute	6
* CyaSSL - 2.7.0	Jun 2013	51,786	33	32 (504)	$\sim$ 2 Minutes	≤ 26	$\leq$ 1 minute	7
wolfSSL - 3.6.6	Aug 2015	103,690	40	256 (31409)	$\sim$ 1 Hour	≤ 26	$\leq$ 1 minute	2
tropicSSL - (Github)	Mar 2013	13,610	66	16 (67)	$\sim$ 1 Minute	≤ 30	$\leq$ 1 minute	10
* PolarSSL - 1.2.8	Jun 2013	29,470	66	56 (90)	$\sim$ 1 Minute	≤ 81	$\leq$ 1 minute	4
mbedTLS - 2.1.4	Jan 2016	53,433	15	13 (536)	$\sim$ 1 Minute	≤ 41	$\leq$ 1 minute	1
* MatrixSSL - 3.4.2	Feb 2013	18,360	9	8 (160)	$\sim$ 1 Minute	1	$\leq$ 1 minute	6
MatrixSSL - 3.7.2	Apr 2015	37,879	30	3240 (8786)	$\sim$ 1 Hour	≤ 25	$\leq$ 1 minute	5

§ The fourth column of the table refers to the lines of code we added to the libraries to make them amenable to our analysis. The fifth and sixth columns display the number of accepting (rejecting) paths we obtained when we made the fields in equivalence class EqC_1 symbolic, and the time it took to complete the extraction process, respectively. The seventh and eighth columns show the upper bound of total paths (including both accepting and rejecting) we observed when the fields in EqC_2 are made symbolic, and the time it took for the path extraction process to complete, respectively.

that it would be difficult for concrete test cases to hit this bug, as the execution is likely to fall through without triggering any noticeable crashes.

Listing 1. Extension Processing In CyaSSL 2.7.0

```
switch (oid) {
...
case AUTH_INFO_OID:
DecodeAuthInfo(&input[idx], length, cert);
break;
case ALT_NAMES_OID:
DecodeAltNames(&input[idx], length, cert);
case AUTH_KEY_OID:
DecodeAuthKeyId(&input[idx], length, cert);
break;
... }
```

C. Findings From Simple Search of Path Constraints

Fields of certificates, represented by symbolic variables in our approach, will appear on path constraints if they are involved in branching decisions either directly or indirectly (*e.g.* some other decision variables were calculated based on their values). Consequently, the second opportunity our approach offers is that immediately after extracting path constraints using symbolic execution, missing checks of fields can be discovered by performing "`grep`" on the path constraints.

Finding 2 *(*pathLenConstraint *ignored in CyaSSL 2.7.0, wolfSSL 3.6.6* [2]*):* We noticed that both of the aforementioned libraries fail to take pathLenConstraint into consideration, which means any such restrictions imposed by upper level issuing CAs would be ignored by the libraries.

This was not reported in [43], where fuzzing was applied to CyaSSL 2.7.0. Interestingly, [43] instead reported that CyaSSL 2.7.0 incorrectly rejects leaf CA certificates given the intermediate CA certificate has a pathLenConstraint of 0, and is noncompliant because such certificates should be accepted according to the RFC. Our findings, however, demonstrate that CyaSSL 2.7.0 could not possibly be rejecting certificates for such a reason because it completely ignores pathLenConstraint. Testing CyaSSL 2.7.0 with concrete certificates confirmed our finding. Thus, the conclusion in [43] that CyaSSL 2.7.0 misinterprets RFC regarding pathLenConstraint and leaf CA certificate is incorrect. We conjecture that this is because the frankencerts used as evidence for such conclusion also happen to contain other errors, and were thus rejected by CyaSSL 2.7.0. This demonstrates the difficulty of interpreting results obtained from fuzzing.

Finding 3 *(*pathLenConstraint *of intermediate CA certificates ignored in tropicSSL, PolarSSL 1.2.8* [3]*):* Our path constraints show that even though both tropicSSL and PolarSSL 1.2.8 recognize the pathLenConstraint variable during parsing time, they check only the one that is on the trusted root certificate during chain validation, and ignores those that are on intermediate CA certificates of a given chain.

In addition to the fact that PolarSSL 1.2.8 does not check pathLenConstraint on intermediate CA certificates, another simple search found that PolarSSL 1.2.8 does not check whether the leaf certificate is CA or not (which is not a noncompliant behavior). It was however reported in [43] that PolarSSL 1.2.8 violates the RFC by always rejecting leaf CA certificates if the intermediate CA certificate has a pathLenConstraint of 0. This is incorrect because PolarSSL 1.2.8 checks neither pathLenConstraint on intermediate CA certificates, nor whether the leaf certificate is CA or not.

Finding 4 *(Certain attribute types of distinguished names ignored in axTLS 1.4.3 and 1.5.3):* Both axTLS 1.4.3 and 1.5.3 ignore the country, state/province and locality attribute types of the issuer and subject names. In other words, organizations from different countries and states having the same name would be considered equivalent during matching. This is a clear deviation from RFC 5280 (Section 4.1.2.4) [2].

We have this finding reported to the developer of axTLS, who acknowledged the existence of the problem and implemented a fix in the new 2.1.1 release.

Finding 5 *(Inability to process* GeneralizedTime *in axTLS 1.4.3, tropicSSL):* RFC 5280 (Section 4.1.2.5) [2] states "Conforming applications MUST be able to process validity dates that are encoded in either UTCTime or GeneralizedTime." However, given our SymCerts with GeneralizedTime, both

[2]wolfSSL 3.9.10 has implemented support for pathLenConstraint [94].

[3]The enforcement of pathLenConstraint from intermediate CA certificates has been introduced since PolarSSL 1.2.18 [95].

tropicSSL and axTLS 1.4.3 returned only 1 concrete rejecting path with an empty path constraint, hence we conclude that the aforementioned libraries cannot handle GeneralizedTime, which is a non-conformance to the RFC. However, the same SymCerts managed to yield meaningful path constraints in axTLS 1.5.3, showing that support for GeneralizedTime has been added in the newer version of axTLS.

Finding 6 *(*KeyUsage *and* ExtKeyUsage *being ignored in MatrixSSL 3.4.2, CyaSSL 2.7.0, tropicSSL):* The three aforementioned implementations do not check KeyUsage and ExtKeyUsage extensions. This noncompliance implies that certificates issued specifically for certain intended purposes (*e.g.* only for software code signing) can be used to authenticate a server in SSL/TLS handshakes. Honoring such restrictions imposed by issuing CAs allows the PKI to implement different levels of trust, and help avoid certificate (and CA) misuse in general.

Finding 7 *(*notBefore *ignored in tropicSSL, PolarSSL 1.2.8;* validity *not checked in MatrixSSL 3.4.2):* Our SymCerts revealed that PolarSSL 1.2.8 does not check the notBefore field, and MatrixSSL 3.4.2 does not have an inbuilt validity check, as there is only 1 path, which is an accepting path with empty constraints, for each of the aforementioned libraries in their respective cases. This is coherent with the findings in [43]. MatrixSSL 3.4.2 delegates the task of checking certificate validity to application developers. tropicSSL has the same problem as PolarSSL 1.2.8, which is not a surprise considering the fact that tropicSSL is a fork of PolarSSL.

Finding 8 *(hhmmss of* UTCTime *ignored in tropicSSL, axTLS 1.4.3 and 1.5.3; hhmmss of both* UTCTime *and* GeneralizedTime *ignored in MatrixSSL 3.7.2):* Given UTCTime on certificates, even though axTLS 1.4.3 and 1.5.3 check for both notBefore and notAfter, they do not take the hour, minute and second into consideration, which means that there could be a shift for as long as a day in terms of rejecting future and expired certificates. This finding is particularly interesting for axTLS 1.5.3, as its implementation of GeneralizedTime support can actually handle hour, minute and second, but for some reason UTCTime is processed in a laxer manner. Following our report, the developer of axTLS has acknowledged the problem and is currently considering a fix. Our extracted path constraints show and tropicSSL also suffer from the same problem.

Unlike its older counterpart, MatrixSSL 3.7.2 has implemented validity checks that handle both UTCTime and GeneralizedTime. However, our extracted path constraints revealed that MatrixSSL 3.7.2 does not attempt to check the time portion of the validity fields, regardless of whether the date-time information is in UTCTime or GeneralizedTime. The developers of MatrixSSL had explained to us the decision to ignore the time portion was made due to its embedded origin, where a local timer might not always be available, and in their own words "*having date set correctly is difficult enough*". They have also admitted that as the result of such decision, a 24-hour shift in rejecting future and expired certificates is inevitable.

Finding 9 *(*notAfter *check applies only to leaf certificate in tropicSSL):* Not just that future certificates are not rejected (e.g. missing check for notBefore as described above) in tropicSSL, our path constraints show that, given a chain of certificates, the check on notAfter only applies to the leaf one. This could lead to severe problems, for instance, if a retired private key of an intermediate issuing CA corresponding to an expired certificate got leaked, attackers would be able to issue new certificates and construct a new chain of certificate that will be accepted by tropicSSL.

Finding 10 *(Incorrect CA certificate and version number assumptions in axTLS 1.4.3 and 1.5.3, CyaSSL 2.7.0, MatrixSSL 3.4.2):* The aforementioned implementations deviate from the RFC in how they establish whether certificates of various versions are CA certificates or not. As explained previously in Section III-A2, in case the certificate has a version older than 3, some out-of-band mechanisms would be necessary to verify whether it is a CA certificate or not. axTLS 1.4.3 and 1.5.3 assume certificates to be CA certificates regardless of the version number. CyaSSL 2.7.0 also does not check the version number, though whenever the basicConstraints extension is present, it will be used to determine whether the certificate is a CA certificate or not. MatrixSSL 3.4.2 does check the version number, and would check the basicConstraints extension for version 3 certificates. However, it would just assume certificates older than version 3 to be CA certificates. The findings on CyaSSL 2.7.0 and MatrixSSL 3.4.2 are coherent with the relevant results reported in [43].

Finding 11 *(Unrecognized critical extensions in MatrixSSL 3.4.2, CyaSSL 2.7.0, axTLS 1.4.3 and 1.5.3):* Section 4.2 of RFC 5280 states "A certificate-using system MUST reject the certificate if it encounters a critical extension it does not recognize or a critical extension that contains information that it cannot process." [2]. Not rejecting unknown critical extensions could lead to interoperability issues. For example, certain entities might define and issue certificates with additional non-standard custom extensions, and rely on the default rejection behavior as described in RFC 5280 to make sure that only a specific group of implementations can handle and process their certificates. However, we found that MatrixSSL v3.4.2 and CyaSSL 2.7.0 would accept certificates with unrecognized critical extensions, which is consistent to the findings in [43].

In addition, we found that axTLS 1.4.3 and 1.5.3 would also accept certificates with unrecognized critical extensions. In fact, based on the path constraints we have extracted, they do not recognize any of the standard extensions that we wanted to test at all, which deviates from RFC 5280, as Section 4.2 says the minimum requirement for applications conforming to the document MUST recognize extensions like key usage, basic constraints, name constraints, and extended key usage, etc. Similarly for mbedTLS 2.1.4, as we have noticed for not implementing support for the name constraints extension, is also noncompliant in that sense. The implication of this is that restrictions imposed by issuing CAs in the form of name constraints will not be honored by mbedTLS 2.1.4, resulting in potential erroneous acceptance of certificates. At

the time of writing, developers of mbedTLS have indicated that they currently have no plans on implementing support for this extension, and suggested that application developers can implement their own if desired.

D. Findings From Cross-Validating Libraries

The final opportunity would be to cross-validate libraries, specifically, for each accepting path of library A and each rejecting path of library B, we perform a conjunction and see if the resulting constraints would be solvable or not. If yes, it signifies a discrepancy exists between the two libraries.

Finding 12 *(*ExtKeyUsage *OID handling in wolfSSL 3.6.6, MatrixSSL 3.7.4):* Our path constraints also unveiled that despite being two of the few libraries that support the extended key usage extension, both wolfSSL 3.6.6 and MatrixSSL 3.7.2 opted for a somewhat lax shortcut in handling the extension: given the object identifier (OID) of a key usage purpose, they do a simple summation (referred colloquially as a non-cryptographic digest function by the developers of MatrixSSL) over all nodes of the OID, and then try to match only that sum. For example, under such scheme, the standard usage purpose "server authentication" (OID 1.3.6.1.5.5.7.3.1, DER-encoded byte values are 0x2B 0x06 0x01 0x05 0x05 0x07 0x03 0x01) would be treated as decimal 71.

Notice that the extension itself is not restricted to only hold standard usage purposes that are defined in the RFC, and custom key usage purposes are common [4]. Since OIDs are only meant to be unique in a hierarchical manner, the sums over nodes of OIDs are not necessarily unique. Hypothetically some enterprises under the private enterprise arc (1.3.6.1.4.1) could define OIDs to describe their own key usage purposes, and if added to the extension, those OIDs might be incorrectly treated as some of the standard key usage purposes by the two libraries. This could be problematic for both interoperability and security, as custom key usage purposes would be misinterpreted, and the standard ones could be spoofed.

This finding is a good example of how our approach can be used to discover the exact treatments that variables undergo inside the libraries during execution. It might also be difficult for unguided fuzzing to hit this particular problem.

We contacted the corresponding developers of the 2 libraries regarding this, and both acknowledged the problem exists. wolfSSL has introduced a more rigorous OID bytes checking since version 3.7.3 [5], and MatrixSSL is planning to incorporate additional checks of the OID bytes in a new release.

Finding 13 *(Incorrect interpretation of* UTCTime *year in MatrixSSL 3.7.2, axTLS 1.4.3 and 1.5.3, tropicSSL):* Since UTCTime reserves only two bytes for representing the year, one needs to be cautious when interpreting it. RFC 5280 Section 4.1.2.5.1 [2] says that when the YY of a UTCTime is larger than or equal to 50 then it should be treated as 19YY, otherwise it should be treated as 20YY. This essentially means that the represented range of year is 1950 to 2049 inclusively.

During cross-validation, we noticed that in certain libraries, some legitimate years are being incorrectly rejected (and accepted). A quick inspection of the path constraints, concrete-value counterexamples, and finally the source code, found the following instances of noncompliance.

Listing 2. UTCTime year adjustment in MatrixSSL 3.7.2

```
y =  2000 + 10 * (c[0] - '0') + (c[1] - '0'); c += 2;
/* Years from '96 through '99 are in the 1900's */
if (y >= 2096) { y -= 100; }
```

As shown in Listing 2, MatrixSSL 3.7.2 interprets any YY less than 96 to be in the twenty first century. This means certificates that had expired back in 1995 would be considered valid, as the expiration date is incorrectly interpreted to be in 2095. On the other hand, long-living certificates that have a validity period began in 1995 would be treated as not valid yet. The developers acknowledged our report on this and have since implemented a fix in a new release.

Listing 3. UTCTime year adjustment in tropicSSL

```
to->year += 100 * (to->year < 90);
to->year += 1900;
```

A similar instance of noncompliance was found in tropicSSL, as shown in Listing 3. tropicSSL interprets any YY less than 90 to be in the twenty first century.

Listing 4. UTCTime year adjustment in axTLS 1.4.3 and 1.5.3

```
if (tm.tm_year <= 50) {    /* 1951-2050 thing */
tm.tm_year += 100;     }
```

A similar issue exists in both axTLS 1.4.3 and 1.5.3. As shown in Listing 4, there is an off-by-one error in the condition used to decide whether to adjust the year or not. In this case, the year 1950 would be incorrectly considered to mean 2050. Based on the inline comment, it seems to be a case where the developer misinterpreted the RFC. A fix has been implemented in a new version of axTLS following our report.

Finding 14 *(Incorrect timezone adjustment in MatrixSSL 3.7.2):* During cross-validation with other libraries, we noticed that the boundary of date checking in the path constraints of MatrixSSL 3.7.2 was shifted by one day. A quick inspection of the date time checking code found that MatrixSSL 3.7.2 uses the `localtime_r()` instead of `gmtime_r()` to convert the current integer epoch time into a time structure. The shift was due to the fact that in conventional `libc` implementations, `localtime_r()` would adjust for the local time zone, which might not necessarily be Zulu, hence deviating from the RFC requirements.

Assuming the date time on certificates are in the Zulu timezone, the implication of this subtle issue is that for systems in GMT-minus time-zones, expired certificates could be considered still valid because of the shift, and certificates that just became valid could be considered not yet valid. Similarly, for systems in GMT-plus time-zones, certificates that are still valid might be considered expired, and future certificates that are not yet valid would be considered valid.

[4]For example, Microsoft defines its own key usage purposes and the corresponding OIDs that are deemed meaningful to the Windows ecosystem [96] (the extension is referred to as "Application Policy" in Microsoft terminology, and is not to be confused with "Certificate Policy").

[5]https://github.com/wolfSSL/wolfssl/commit/d248a7660cc441b68dc48728b10256e852928ea3

We discussed this with the developers of MatrixSSL. They conjectured the reason for using `localtime_r()` instead of `gmtime_r()` was due to the latter being unavailable on certain embedded platforms. They have agreed, however, as MatrixSSL is gaining popularity on non-embedded platforms, in a new release, they will start using `gmtime_r()` on platforms that support it.

Finding 15 *(Overly restrictive* notBefore *check in CyaSSL 2.7.0* [6]*):* RFC 5280 Section 4.1.2.5 says "The validity period for a certificate is the period of time from notBefore through notAfter, inclusive." However, when cross-validating CyaSSL 2.7.0 with other libraries, from the concrete counterexamples we noticed that discrepancy exists in how the same notBefore values would be accepted by other libraries but rejected by CyaSSL 2.7.0, while such discrepancy was not observed with notAfter. An inspection of the notBefore checking code yielded the following instance of noncompliance:

Listing 5. Erroneous "less than" check in CyaSSL 2.7.0

```
static INLINE int DateLessThan(const struct tm* a, const
    struct tm* b)
{ return !DateGreaterThan(a,b); }
```

Notice that the negation of $>$ is $\leq$, not $<$, which explains why if the current date time happen to be the same as the one described in notBefore, the certificate would be considered future (not valid yet) and rejected. Hence the notBefore checking in CyaSSL 2.7.0 turns out to be overly restrictive than what the RFC mandates.

This is again a new result, comparing to the previous work [43] that also studied CyaSSL 2.7.0. Our conjecture is that given a large number of possible values, it might be difficult for unguided fuzzing to hit boundary cases, hence such a subtle logical error eluded their analysis.

Finding 16 *(*KeyUsage *and* ExtKeyUsage *being ignored in PolarSSL 1.2.8):* The fact that PolarSSL 1.2.8 does not check KeyUsage and ExtKeyUsage, evaded our simple search approach but was caught during cross-validation, as the implementation actually parses the two extensions, hence some constraints were added as the result of several basic sanity checks happened during parsing. However, during cross-validation, it became clear that apart from the parsing sanity checks, PolarSSL 1.2.8 does not do any meaningful checks on KeyUsage and ExtKeyUsage.

In fact, this resulted in another instance of noncompliance, as PolarSSL 1.2.8 would not reject certificates with KeyUsage or ExtKeyUsage, even if those two extensions were made critical, and it does not perform any meaningful checks apart from merely parsing them. This is an example where a library is intended to handle an extension but was not able to, because of incomplete implementation.

This is consistent with similar results reported in [43], although the finding that PolarSSL 1.2.8 does not check the KeyUsage extension on intermediate CA certificates was not reported in that paper.

[6]This has been fixed in newer versions of CyaSSL and WolfSSL.

Finding 17 *(*pathLenConstraint *of trusted root misinterpreted in tropicSSL):* During cross validation, it became clear to us that, in tropicSSL: (1) on one hand, some accepting paths would allow the pathLenConstraint variable to be 0; (2) on the other hand, some rejecting paths reject because the pathLenConstraint was deemed to be smaller than an unexpectedly large boundary. In both cases, the pathLenConstraint variable appears to have been misinterpreted by tropicSSL.

We suspect that this might be due to the value 0 in the internal parsed certificate data structure is used to capture the case where the pathLenConstraint variable is absent (*i.e.* no limit is imposed). A quick inspection of the parsing code revealed that our suspicion is indeed correct. In fact, the parsing code is supposed to always add 1 to the variable if it is present on the certificate, but a coding error [7] of missing a dereferencing operator ($\star$) in front of an integer pointer means that the increment was applied to the pointer itself but not the value, hence the observed behavior described above.

This subtle bug has a severe implication: it completely defeats the purpose of imposing such restriction on a certificate, as a pathLenConstraint of 0 would be incorrectly treated to mean that the chain length could be unlimited.

Finding 18 *(Not critical means not a CA in tropicSSL):* During cross validation, we also noticed that when the intermediate CA certificate's basicConstraints extension is set to non-critical, and the isCA boolean is set to `True`, tropicSSL would consider the intermediate CA certificate not a CA certificate. Additionally, in the path constraints, the symbolic variable representing the criticality of basicConstraints and the one that represents the isCA boolean are always in conjunction through a logical `AND`.

A quick inspection found the following problem in the parsing code that handles the basicConstraints extension:

Listing 6. Incorrect adjustment to the isCA boolean in tropicSSL

```
*ca_istrue = is_critical & is_cacert;
```

This interpretation of the basicConstraints extension deviates from the specification, as RFC 5280 says that clients should process extensions that they can recognize, regardless of whether the extension is critical or not. The criticality of basicConstraints should not affect the semantic meaning of attributes in the extension itself. This is an example of a CCVL being overly restrictive.

E. Other findings

Here we present other interesting findings that are not explicitly noncompliant behaviors deviating from RFC 5280.

Extra 1 *(Ineffective date string sanity check in MatrixSSL 3.7.2):* During cross-validation, we noticed that date time byte values in MatrixSSL 3.7.2 are not bounded for exceedingly large or unexpectedly small values. However, in the constraints, we see combinations of whether each byte is too small or not (though not affecting the acceptance decision), which looked suspiciously like a failed lower boundary check. A quick inspection of the certificate parsing code unveiled the

[7]This has been fixed in later versions of PolarSSL and mbedTLS.

snippet shown in Listing 7 that is meant to vet a given date string from a certificate, and reject it with a parser error if the values are outside of an expected range. Unfortunately, due to incorrectly using the `&&` operator instead of `||`, the `if` conditions are never satisfiable. This is also proven by the fact that if we symbolically execute the code snippet in Listing 7, all possible execution paths returns 1. Consequently that code snippet would actually never reject any given strings, hence completely defeating the purpose of having a sanity check.

Listing 7. date string sanity check in MatrixSSL 3.7.2

```
if (utctime != 1) {    /* 4 character year */
if (*c < '1' && *c > '2') return 0; c++; /* Year */
if (*c < '0' && *c > '9') return 0; c++;
}
if (*c < '0' && *c > '9') return 0; c++;
if (*c < '0' && *c > '9') return 0; c++;
if (*c < '0' && *c > '1') return 0; c++;    /* Month */
if (*c < '0' && *c > '9') return 0; c++;
if (*c < '0' && *c > '3') return 0; c++;    /* Day */
if (*c < '0' && *c > '9') return 0;
return 1;
```

Following our report, the developers of MatrixSSL have acknowledged this is indeed a faulty implementation. Along with other fixes being implemented to make date-time processing more robust, they have decided that this sanity check will no longer be used in newer versions of MatrixSSL.

Extra 2 *(*notBefore *and* notAfter *bytes taken "as is" in CyaSSL 2.7.0, WolfSSL 3.6.6, axTLS 1.4.3 and 1.5.3):* For the four aforementioned implementations, we noticed during cross-validation that they do not perform any explicit boundary checks on the value of the date time value bytes of notBefore and notAfter, and just assumed that those bytes are going to be valid ASCII digits (*i.e.* 0–9). It is hence possible to put other ASCII characters in the date time bytes and obtain an exceptionally large (small) values for notAfter (notBefore), though this does not seem to be an imminent threat, nor does it violate the RFC, as the RFC did not stipulate what implementations should do.

Extra 3 *(Timezone Handling):* Another discrepancy that we have observed during cross-validating path constraints of different libraries was how they impose/assume the time zone of notBefore and notAfter on certificates. Specifically, we notice that mbedTLS 2.1.4 and wolfSSL v2.3.3 would reject certificates that do not have the timezone ending with a 'Z'.

This is possibly due to the fact that RFC 5280 [2] mandates conforming CAs to express validity in Zulu time (a.k.a GMT or Zero Meridian Time) when issuing certificates, regardless of the type being UTCTime or GeneralizedTime. Other implementations like MatrixSSL 3.7.2, axTLS 1.5.3 and PolarSSL 1.2.8 ignore the timezone character and simply assume the Zulu timezone is always being used.

This is arguably an example of under-specification, as it is not clear whether implementations should try to handle (with proper time zone adjustment) or reject certificates with a non-Zulu timezone, since RFC 5280 [2] did not explicitly mandate an expected behavior.

VII. Discussion

A. Takeaway for Application Developers

As a takeaway for application developers that need to use SSL/TLS libraries for processing X.509 certificates, a general rule of thumb is to upgrade to newer versions of the libraries if possible. As demonstrated by our findings, newer versions of implementations, even when originated from the same source tree as their legacy counterparts, are better equipped in terms of features and extension handling, as well as in general having more rigorous checks. Holding on to legacy code could potentially hurt both security and interoperability. Unfortunately, regular software patching, particularly for IoT devices, does not seem to happen widespread enough [97].

We understand that due to the needs to optimize for different application scenarios (*e.g.* small footprint for resource constrained platforms), certain features might not be implemented in their entirety as described in the standard specifications. In order to help application developers to better understand the trade-offs and make a more well-informed decision in choosing which SSL/TLS library to use, we believe that one possibility would be to have a certification program that tests for implementation conformance and interoperability, similar to that of the IPv6 Ready Logo Program [98], and the High Definition Logos [99]. For example, an "X.509 Gold" for libraries that implement most required features correctly, and an "X.509 Ready" for libraries that can only handle the bare minimum but are missing out on certain features.

B. Limitations

Since our noncompliance detection approach critically relies on symbolic execution which is known to suffer from path explosion, especially in the presence of symbolic data-dependent loops, it is deliberately made to **trade away completeness for practicality** (*i.e.*, our approach is not guaranteed to reveal all possible noncompliances in an implementation and can have false negatives).

Our current scope of analysis does not include the logic for checking certification revocation status and hostname matching. As noted in [43], for both revocation status checking and hostname matching, while some libraries provide relevant facilities, some delegate the task to application developers. In addition, a typical implementation of a hostname matching logic uses complex string operations and analyzing these require a dedicated SMT solver with support for the theory of strings [100]. We leave that for future work.

Moreover, as we use concrete values in SymCerts, symbolic execution sway away from rigorously exercising the parsing logic. Though we have uncovered parsing bugs as reported in Section VI, our scrutiny on the parsing code is not meant to be comprehensive. Noticeably, low-level memory errors due to incorrect buffer management in the parsing code, as reported in a recent Vulnerability Note [101], can elude our analysis.

C. Threat to Validity

In some cases during certificate validation, it is not clear who is required to perform the validity check on a field, *i.e.*,

the underlying library or the application using the library. The RFC states that some specific validity check must be performed without clearly identifying the responsible party. This unclear separation of responsibilities have resulted in libraries opting for significantly different API designs. We rely on example usage—often come with the source code in the form of a sample client—to draw a boundary for extracting the approximated certificate accepting (and rejecting) universes. Optional function calls to extra checking logics, if not demonstrated in the sample client programs, will be missed by our analysis. Additionally, if some of the checks performed on certificates are being pushed down to a different phase during SSL/TLS handshake instead of the server certificate validation phase, these checks might be missing from our extraction. We rely on the concrete client-server replay setup to catch them and iteratively include them in the extraction.

Our optimization often rely on the expectation that the value of *some* fields are handled in the implementation in an uniform way. For checking validity of fields that can have variable lengths, we assume the implementation treats each regular length (**not corner cases**) uniformly. In addition, we also assume that the semantic independence of certain certificate fields are maintained in the implementation. For instance, we assume that the certificate validity fields are not dependent on any other fields. Although we have observed that this seems to be the case and the RFC supports it, hypothetically a developer can mistakenly create an artificial dependency.

VIII. Conclusion and Future Direction

In this paper, we present a novel approach that leverages symbolic execution to find noncompliance in X.509 implementations. In alignment with the general consensus, we observe that due to the recursive nature of certificate representation, an off-the-shelf symbolic execution engine suffers from path explosion problem. We overcome this inherent challenge in two ways: (1) Focusing on real implementations with a small resource footprint; (2) Leveraging domain-specific insights, abstractions, and compartmentalization. We use SymCerts—certificate chains in which each certificate has a mix of symbolic and concrete values—such that symbolic execution can be made scalable on many X.509 implementations while meaningful analysis can be conducted.

We applied our noncompliance approach to analyze 9 real implementations selected from 4 major families of SSL/TLS source base. Our analysis exposed 48 instances of noncompliance, some of which has severe security implications. We have responsibly shared our new findings with the respective library developers. Most of our reports have generated positive acknowledgments from the developers, and led to the implementation of fixes to the said problems in new releases.
We now identify possible ways of extending our work.

The Vision of Fully Automatic Noncompliance Detection: Our current approach is a significant first step towards a fully automatic noncompliance detection approach for X.509 CCVL implementation. We envision the following two automatic approaches that can leverage our current work.

(1) *Signature-driven Automated Noncompliance Detection:* Vulnerability signatures capture properties that should not hold for any CCVL implementations. If one can represent a vulnerability signature as a QFFOL formula Ψ, then for a given implementation I, we can check, using an SMT solver, whether there exists a certificate in I's certificate accepting universe that satisfies Ψ.

(2) *Cross Validation With a Reference Implementation:* For this approach, one would need to develop a formally verified reference implementation of X.509 CCVL, I_{ref}. Such a reference implementation can be highly valuable and is also sought after by the research community, as indicated by a panel in a NSF workshop titled "Formal Methods for Security" [102]. For automatically detecting noncompliance of a given implementation I_{test}, one can simply cross validate I_{test} against I_{ref} using our automatic cross validation approach.

Analyzing Libraries for Conventional Systems: SSL/TLS libraries that are developed for traditional systems (*e.g.*, OpenSSL, GnuTLS, Mozilla NSS) are not included in our current analysis. We leave the analysis of these libraries as a subject of future work.

Acknowledgment

We thank the various developers of the SSL/TLS libraries for their efforts in implementing and maintaining all the tested open source implementations. In particular, we sincerely appreciate the developers of WolfSSL, MatrixSSL and axTLS, for giving us prompt feedbacks and implementing new fixes in a timely manner. We would also like to thank the anonymous reviewers for their helpful comments. The work reported here serves as the basis for the NSF grant CNS-1657124; it was supported in part by a Purdue Research Foundation award, and the NSF grants CNS-1314688 and CNS-1421815.

References

[1] ITU-T Recommendation X.509 (2005) — ISO/IEC 9594-8:2005, "Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks," *International Telecommunication Union*, 2005.

[2] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," Internet Requests for Comments, Tech. Rep. 5280, May 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt

[3] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *IEEE Symposium on Security and Privacy*, 2015.

[4] J. De Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 193–206.

[5] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1492–1504.

[6] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell, "Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 223–238.

[7] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 445–459.

[8] B. Beurdouche, A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and K. Bhargavan, "FLEXTLS: A Tool for Testing TLS Implementations," in *Proceedings of the 9th USENIX Conference on Offensive Technologies*, ser. WOOT'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831211.2831212

[9] "CVE-2016-1115," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1115.

[10] "CVE-2016-5669," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-5669.

[11] "CVE-2016-5672," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-5672.

[12] "CVE-2016-2180," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2180.

[13] "CVE-2016-5655," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-5655.

[14] "CVE-2016-3664," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-3664.

[15] "CVE-2016-2113," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2113.

[16] "CVE-2016-1563," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1563.

[17] "CVE-2016-2562," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2562.

[18] "CVE-2016-2047," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2047.

[19] "CVE-2015-5655," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-5655.

[20] "CVE-2014-0092," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0092.

[21] "CVE-2014-1266," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266.

[22] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 2, no. 42, pp. 230–265, 1936.

[23] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, M. Roberts, S. Setty, and B. Zill, "Ironfleet: Proving practical distributed systems correct," in *SOSP*, 2015.

[24] K. L. McMillan, *Symbolic model checking*. Springer, 1993.

[25] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of c programs," in *ACM SIGPLAN Notices*, vol. 36, no. 5. ACM, 2001, pp. 203–213.

[26] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, "Zing: A model checker for concurrent software," in *CAV*, 2004.

[27] G. Holzmann and M. Smith, "A practical method for verifying event-driven software," in *ICSE*. ACM, 1999.

[28] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 21, 2009.

[29] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 58–70.

[30] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, 2000, pp. 154–169.

[31] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "Synergy: A new algorithm for property checking," in *FSE*, 2006.

[32] T. Ball and S. Rajamani, "The slam project: Debugging system software via static analysis," *SIGPLAN Not.*, vol. 37, no. 1, 2002.

[33] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast: Applications to software engineering," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5, 2007.

[34] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 388–402, 2004.

[35] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Satabs: Sat-based predicate abstraction for ansi-c," in *TACAS*. Springer-Verlag, 2005.

[36] J. Esparza, S. Kiefer, and S. Schwoon, "Abstraction refinement with Craig interpolation and symbolic pushdown systems," in *TACAS*. Springer, 2006.

[37] S. Löwe, "Cpachecker with explicit-value analysis based on cegar and interpolation," in *TACAS*. Springer-Verlag, 2013.

[38] G. Brat, K. Havelund, S. Park, and W. Visser, "Model checking programs," in *IEEE International Conference on Automated Software Engineering (ASE)*. Citeseer, 2000, pp. 3–12.

[39] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. New York, NY, USA: Cambridge University Press, 1993.

[40] J. Jaffar, V. Murali, J. Navas, and A. Santosa, "Tracer: A symbolic execution tool for verification," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, P. Madhusudan and S. Seshia, Eds. Springer Berlin Heidelberg, 2012, vol. 7358, pp. 758–766.

[41] E. M. Clarke, S. Jha, and W. Marrero, "Verifying security protocols with brutus," *TOSEM*, vol. 9, no. 4, 2000.

[42] P. Godefroid, "Model checking for programming languages using verisoft," in *POPL*. ACM, 1997, pp. 174–186.

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

[44] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[45] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008, pp. 209–224.

[46] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *ICSE*, 2011, pp. 1066–1071.

[47] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[48] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254088

[49] *HTTPS client is here for the Photon!*, 2015 (accessed Nov 3, 2016), https://community.particle.io/t/https-client-is-here-for-the-photon-by-the-glowfi-sh-team/15934.

[50] *spark / firmware / communication / lib*, 2016 (accessed Nov 3, 2016), https://github.com/spark/firmware/tree/master/communication/lib.

[51] *Arduino/libraries/ESP8266WiFi/src/include/ssl.h*, 2016 (accessed Feb 2, 2017), https://github.com/esp8266/Arduino/blob/master/libraries/ESP8266WiFi/src/include/ssl.h.

[52] *micropython/extmod*, 2017 (accessed Feb 2, 2017), https://github.com/micropython/micropython/tree/master/extmod.

[53] "CVE-2016-6303," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-6303.

[54] "CVE-2016-7052," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-7052.

[55] "CVE-2016-6305," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-6305.

[56] P.-H. Kamp, "Please put openssl out of its misery," *Queue*, vol. 12, no. 3, pp. 20:20–20:23, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602649.2602816

[57] *how to install curl and libcurl*, (accessed Nov 3, 2016), https://curl.haxx.se/docs/install.html.

[58] *ipsvd - internet protocol service daemons - installation*, (accessed Feb 2, 2017), http://smarden.org/ipsvd/install.html.

[59] *Package: gatling (0.12cvs20120114-4) high performance web server and file server*, (accessed Feb 2, 2017), https://packages.debian.org/wheezy/gatling.

[60] *A Python library that encapsulates wolfSSL's wolfCrypt API.*, (accessed Feb 2, 2017), https://pypi.python.org/pypi/wolfcrypt/0.2.0.

[61] *mbed TLS (PolarSSL) wrapper*, (accessed Feb 2, 2017), https://pypi.python.org/pypi/python-mbedtls/0.6.

[62] W. M. McKeeman, "Differential testing for software," *DIGITAL TECHNICAL JOURNAL*, vol. 10, no. 1, pp. 100–107, 1998.

[63] R. B. Evans and A. Savoia, "Differential testing: A new approach to change detection," in *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, ser. ESEC-FSE companion '07. New York, NY, USA: ACM, 2007, pp. 549–552. [Online]. Available: http://doi.acm.org/10.1145/1295014.1295038

[64] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, February 2009, vol. 185, ch. 26, pp. 825–885.

[65] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 519–531.

[66] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson, "Analyzing Forged SSL Certificates in the Wild," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 83–97.

[67] C. Hlauschek, M. Gruber, F. Fankhauser, and C. Schanes, "Prying open Pandora's box: KCI attacks against TLS," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[68] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, K. R. Butler, and A. Alkhelaifi, "Securing ssl certificate verification through dynamic linking," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 394–405.

[69] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 38–49.

[70] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 98–113.

[71] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting ssl usage in applications with sslint," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 519–534.

[72] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing api usages through semantic cross-checking," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 363–378. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun

[73] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," *TISSEC*, vol. 12, no. 2, 2008.

[74] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *ESEC/FSE*. ACM, 2005.

[75] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," in *PLDI*, 2002.

[76] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.

[77] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: a platform for in-vivo multi-path analysis of software systems," in *ASPLOS*, 2012.

[78] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*. ACM, 2005.

[79] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin, "Automating software testing using program analysis," *Software, IEEE*, vol. 25, no. 5, pp. 30–37, 2008.

[80] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 669–685. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032360

[81] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 49–64. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[82] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein, "Analyzing protocol implementations for interoperability," in *NSDI*, 2015.

[83] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *NSDI*, 2012.

[84] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A soft way for openflow switch interoperability testing," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 265–276. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413207

[85] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 361–377. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815422

[86] L. D'Antoni and M. Veanes, "Minimization of symbolic automata," *SIGPLAN Not.*, vol. 49, no. 1, pp. 541–553, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2578855.2535849

[87] G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis, "Back in black: towards formal, black box analysis of sanitizers and filters," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 91–109.

[88] George Argyros and Ioannis Stais and Suman Jana and Angelos D. Keromytis and Aggelos Kiayias, "SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, Oct 2016.

[89] C. Adams and S. Lloyd, *Understanding PKI: Concepts, Standards, and Deployment Considerations*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[90] G. Nelson and D. C. Oppen, "Fast decision procedures based on congruence closure," *J. ACM*, vol. 27, no. 2, pp. 356–364, Apr. 1980.

[91] S. Legg, "ASN.1 Module Definition for the LDAP and X.500 Component Matching Rules," RFC 3727, Mar. 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc3727.txt

[92] C. Gardiner and C. Wallace, "ASN.1 Translation," RFC 6025, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc6025.txt

[93] "ASN.1 JavaScript decoder," https://lapo.it/asn1js/.

[94] *wolfSSL ChangeLog*, 2016 (accessed Oct 22, 2016), https://www.wolfssl.com/wolfSSL/Docs-wolfssl-changelog.html.

[95] *mbed TLS 2.2.0, 2.1.3, 1.3.15 and PolarSSL 1.2.18 released*, 2015 (accessed Mar 14, 2017), https://tls.mbed.org/tech-updates/releases/mbedtls-2.2.0-2.1.3-1.3.15-and-polarssl.1.2.18-released.

[96] *Certificate Template Extensions: Application Policy*, (accessed Oct 28, 2016), https://technet.microsoft.com/en-us/library/cc731792(v=ws.11).aspx.

[97] C. Young, *Flawed MatrixSSL Code Highlights Need for Better IoT Update Practices*, 2016 (accessed Feb 6, 2017), http://www.tripwire.com/state-of-security/security-data-protection/cyber-security/flawed-matrixssl-code-highlights-need-for-better-iot-update-practices/.

[98] *IPv6 Ready Logo Program*, 2016 (accessed Sept 04, 2016), https://www.ipv6ready.org.

[99] *High Definition Logos*, 2016 (accessed Sept 04, 2016), http://www.digitaleurope.org/Services/High-Definition-Logos.

[100] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, *A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions*. Cham: Springer International Publishing, 2014, pp. 646–662. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08867-9_43

[101] *Vulnerability Note VU#396440 - MatrixSSL contains multiple vulnerabilities*, 2016 (accessed Feb 6, 2017), http://www.kb.cert.org/vuls/id/396440.

[102] S. Chong, J. Guttman, A. Datta, A. C. Myers, B. Pierce, P. Schaumont, T. Sherwood, and N. Zeldovich, "Report on the NSF workshop on formal methods for security," *CoRR*, vol. abs/1608.00678, 2016. [Online]. Available: http://arxiv.org/abs/1608.00678