Published: 19 June 2025
Accepted: 14 January 2025
Received: 13 September 2024

RESEARCH-ARTICLE

On the Unnecessary Complexity of Names in X.509 and Their Impact on Implementations

YUTENG SUN, Chinese University of Hong Kong, Hong Kong, Hong Kong

JOYANTA DEBNATH, Stony Brook University, Stony Brook, NY, United States

WENZHENG HONG

OMAR CHOWDHURY, Stony Brook University, Stony Brook, NY, United States

SZE YIU CHAU, Chinese University of Hong Kong, Hong Kong, Hong Kong

On the Unnecessary Complexity of Names in X.509 and Their Impact on Implementations

YUTENG SUN, The Chinese University of Hong Kong, Hong Kong
JOYANTA DEBNATH, Stony Brook University, USA
WENZHENG HONG, Independent, China
OMAR CHOWDHURY, Stony Brook University, USA
SZE YIU CHAU, The Chinese University of Hong Kong, Hong Kong

The X.509 Public Key Infrastructure (PKI) provides a cryptographically verifiable mechanism for authenticating a binding of an entity's public-key with its identity, presented in a tamper-proof digital certificate. This often serves as a foundational building block for achieving different security guarantees in many critical applications and protocols (*e.g.*, SSL/TLS). Identities in the context of X.509 PKI are often captured as names, which are encoded in certificates as composite records with different optional fields that can store various types of string values (*e.g.*, ASCII, UTF8). Although such flexibility allows for the support of diverse name types (*e.g.*, IP addresses, DNS names) and application scenarios, it imposes on library developers obligations to enforce unnecessarily convoluted requirements. Bugs in enforcing these requirements can lead to potential interoperability and performance issues, and might open doors to impersonation attacks. This paper focuses on analyzing how open-source libraries enforce the constraints regarding the formatting, encoding, and processing of complex name structures on X.509 certificate chains, for the purpose of establishing identities. Our analysis reveals that a portfolio of simplistic testing approaches can expose blatant violations of the requirements in widely used open-source libraries. Although there is a substantial amount of prior work that focused on testing the overall certificate chain validation process of X.509 libraries, the identified violations have escaped their scrutiny. To make matters worse, we observe that almost all the analyzed libraries completely ignore certain pre-processing steps prescribed by the standard. This begs the question of whether it is beneficial to have a standard so flexible but complex that none of the implementations can faithfully adhere to it. With our test results, we argue in the negative, and explain how simpler alternatives (*e.g.*, other forms of identifiers such as Authority and Subject Key Identifiers) can be used to enforce similar constraints with no loss of security.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Interoperability*; • **Networks → Security protocols**.

Additional Key Words and Phrases: X.509 Certificates, Identities and names, Compliance with standards

ACM Reference Format:
Yuteng Sun, Joyanta Debnath, Wenzheng Hong, Omar Chowdhury, and Sze Yiu Chau. 2025. On the Unnecessary Complexity of Names in X.509 and Their Impact on Implementations. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE066 (July 2025), 21 pages. https://doi.org/10.1145/3715785

1 Introduction

The X.509 Public-Key Infrastructure (PKI) [31] has found widespread applications as the authentication provider for many different critical security protocols and applications such as Internet

Authors' Contact Information: Yuteng Sun, The Chinese University of Hong Kong, Sha Tin, Hong Kong, sy021@ie.cuhk.edu.hk; Joyanta Debnath, Stony Brook University, Stony Brook, USA, jdebnath@cs.stonybrook.edu; Wenzheng Hong, Independent, Shanghai, China, wzhong20@fudan.edu.cn; Omar Chowdhury, Stony Brook University, Stony Brook, USA, omar@cs.stonybrook.edu; Sze Yiu Chau, The Chinese University of Hong Kong, Sha Tin, Hong Kong, sychau@ie.cuhk.edu.hk.

ACM 2994-970X/2025/7-ARTFSE066
https://doi.org/10.1145/3715785

Protocol Security (IPSec), Transport Layer Security (TLS), and Secure/Multipurpose internet Mail Extensions (S/MIME). At its core, X.509 PKI provides a cryptographically verifiable way for establishing a trusted binding between an entity's public key and its *identities*, which is presented in a tamper-proof container called *digital certificate*. This ability to *authenticate* entities often serves as the foundation for other security guarantees, such as confidentiality and non-repudiation.

Each certificate has an *issuer*, typically a Certificate Authority (CA), and a *subject* (*i.e.*, the owner of the certificate), in which the issuer vouches for the public key-identity binding of the subject by digitally signing the certificate with its own private key. In general, when a communicating peer p_1 wants to authenticate itself to another peer p_2, p_1 presents a chain of certificates including its own certificate as evidence. It is the responsibility of p_2 to establish the authenticity of this evidence, starting from its trust anchors (a collection of CA certificates unconditionally trusted by p_2) to p_1's certificate through the *issuer-subject relationship*. This is typically done in accordance to the algorithm prescribed in RFC 5280, sometimes known as the *X.509 certificate chain validation*. After establishing the chain validity, p_2 then needs to check that the *identities* conveyed on the certificate of p_1 indeed match the expectations of p_2. This is often referred to as *hostname verification* for domain name-based identities.

Conceptually, authenticating an entity with certificates can be decomposed into the following high-level steps: ❶ *parsing* the certificates in the input chain and the system's trust anchors; ❷ *building* candidate certificate chains using certificates present in the input chain and other known certificates, such that the candidate chain starts from a trust anchor and maintains the *issuer-subject relationship* between every two adjacent certificates of the chain; ❸ *verifying* the digital signature of each certificate of the candidate chain using the public-key of its predecessor; ❹ *imposing* additional restrictions stipulated in other certificate fields and extensions; ❺ *checking* whether the identity of the subject in the last certificate of the candidate chain is the same as the expected identity of the peer to be authenticated. While prior work has analyzed in depth steps ❹ [4, 7, 11, 33] and ❸ [8, 24, 36], only limited attempts have been put on investigating step ❺ [32]. Also, the investigations on ❷ so far only concern about a very limited subset of the subject and issuer names outlined in the standard documents [7, 11]. *This paper focuses on analyzing the implementations of steps ❷ and ❺ in open-source libraries.*

Identities play a critical role in the realization of the authentication process described above, especially in steps ❷ and ❺. Identities in this context are represented as *names*. One can possibly have three types of names in a certificate that play a role in the authentication process[1], namely, issuer name, subject name, and the optional subject alternative names (an extension called SAN), which can be viewed as aliases of the subject's name. Step ❷ mainly involves the first two types, whereas step ❺ focuses on SAN (when present) or subject name (typically when SAN is absent). At the core, both steps ❷ and ❺ checks to see whether two names *match*, although the semantics of what is considered a match can be substantially different in both cases. As an example, SAN may have wild card characters which may require matching a string with a regular expression. Also, names might go through different canonicalizations and transformations (*e.g.*, LDAP stringprep for ❷ and Punycode for ❺) before they get matched.

Although the idea of having string-based names to describe the identities bound by certificates is intuitive, correctly implementing steps ❷ and ❺ is non-trivial, in part due to flexible and complexity of how names are formatted and encoded in a certificate. Abstractly, the name fields can be viewed as an unordered sequence of attribute-value pairs. An attribute can be absent, present with meaningful values, or present as an empty string. Depending on the attribute types, letters in such string values can be drawn from different character sets (*e.g.*, subsets of ISO 10646 and ASCII) and encoded

[1]Issuer Alternative Name exists as an extension but is not used in certification path validation (Section 4.2.1.7 of RFC5280).

differently (*e.g.*, UTF-8, UTF-32, ASCII, *etc.*). An overview of the name attributes, string types and character sets (charsets) used in X.509 certificates can be found in Tables 1 and Table 2.

Motivated by some recent vulnerabilities found in name process (*e.g.*, CVE-2022-3602 [13] in OpenSSL) and the research gap in investigating X.509 name checking, in this paper, we focus on the problem of testing and analyzing the name processing and matching logic implemented in cryptographic libraries. For analyzing the implementations, we employ a portfolio of approaches ranging from RFC-guided test case crafting to more automated testing approaches such as adaptive fuzz testing and dynamic symbolic execution. At a high level, we found that none of the libraries faithfully follow the standard requirements on name processing and matching. For example, our testing revealed that none of the libraries implement the LDAP stringprep pre-processing algorithm prescribed for step ❷. Additionally, we found instances of out-of-bounds read and potential name confusions in step ❺ of some libraries. A key observation from our findings is that developers of different libraries often improvise their own ways of processing and matching X.509 names, which do not follow the standards, and can lead to potential interoperability and security problems. Overall, we discovered 119 instances of deviation from standard requirements

We note that the main point of this paper is not about proposing new bug finding tools that is better than existing tools. Nevertheless, in terms of investigating implementations of X.509 name checking, our testing has achieved better coverage than previous work. In fact, it is quite interesting to see that even a portfolio of tried-and-true testing approaches can reveal many deviations from the standards. That said, while some of the bugs appear to be programming blunders, for a large portion of the findings, we do not attribute blame to the developers. Our position is that the standards concerning names in X.509 are unnecessarily complex with no apparent benefits, and we sympathize with the developers for not faithfully implementing all of the standard requirements, many of which are arguably historical bloats. This position is further justified by the observation that logical errors in step ❷ typically only contribute to interoperability and performance issues, but do not hamper overall security, due to the cryptographic guarantees of digital signatures.

All in all, in this paper, we provide ample empirical evidence to show that the complexity of names in X.509 is so high that it actually *deters* compliant and correct implementations. This begs the philosophical question: *if a standard is not followed by anyone, is it still a meaningful standard?* With the lessons learned from our investigation, we answer this question in the negative. In particularly, we argue that for a security-critical infrastructure such as X.509, it is more beneficial to focus on simplicity, instead of carrying over some rarely used flexible options, the benefits of which are often historical and hypothetical. Based on our findings and discussions exchanged with several library developers, we have identified possibilities that, without breaking compatibility with the current X.509 format, can help tame its unnecessary complexity. As an example, the presence of subject and authority key identifiers (known as extensions SKI and AKI) can make step ❷ much simpler by checking equality of identifiers instead of dealing with names. Furthermore, the original X.509 is a general profile that is not tied to a particular application. By focusing on a specific application, which makes sense for important ones such as the Web, stronger assumptions can be brought in by its standardization committees (*e.g.*, the CA/B forum for Web) to further simplify the name format and processing rules, which can ultimately lead to smaller and more consistent implementations.

Our contribution can be summarized as follows:

- We use three testing approaches to comprehensively analyze the name processing and matching logic in 23 open-source X.509 PKI libraries, discovering *119* instances of deviations from standards. This exposes a significant gap between the requirements stipulated in RFCs and actual real-world implementations.

- We evaluate our test coverage and show that our portfolio of testing approaches indeed led to a more in-depth investigation of X.509 name processing than previous work.
- We have responsibly shared our findings with library developers, and exchanged opinions with them. At the time of writing, two CVE numbers are assigned, and several libraries have already made enhancements.
- Drawing from the lessons learned from our findings and developers' opinions, we discuss practical ways of taming the complexity of names in X.509.

2 Preliminaries

This section introduces the X.509 certificate structure and the verification process. We briefly explain the role that name strings play in certificate verification, the relevant string processing rules, and the notation we use. An overview of the name-related verification tasks, *i.e.*, name chaining (step ❷) and hostname verification (step ❺), as well as the relevant fields and extensions can be found in Figure 1.

2.1 Name Attributes and String Types

There are many name attributes defined for X.509. The common ones include *[CN]*, which refers to the common name attribute in issuer/subject name, and *[dns]*, which refers to the `dnsName` attribute in `SAN`. A list of commonly used name attributes is shown in Table 1.

Name attributes can take on various string types, the options of which are shown in Table 2. Note that these string types take different character sets (charsets) and encoding rules. The hierarchy of their charsets is visualized in Figure 2. Also note that even if 2 string types take the same charset, their encoding rules can still be different. For example, both *utf8String* and *universalString* take the Unicode charset, but *universalString* uses 4 bytes to encode a character, while *utf8String* has variable-length encoding (1-4 bytes). As we will show later, these string types, charsets, and encoding are a major source of confusion and standard deviation in X.509 implementations.

Notation. In the rest of the paper, we will use the notation

[attrName]=<strType>'strVal'(bytes)

to denote a name attribute string. As an example, *[CN]='aa'(0x6161)* describes a common name of string type *utf8String*, with the string value being 'aa'. For simplicity, *(bytes)* will not be shown if the *'strVal'* is self-evident. We use *(bytes) = (None)* to denote an empty string. If *<strType>* is not present, the default string type for *[CN]* and *[dns]* are *utf8String* and *ia5String*, respectively. We also define a special attribute, *[in]*, which is used to denote the (expected) hostname input that an application programmer has provided for hostname verification, which takes the string type *ia5String*.

2.2 Name Chaining (ℕℂ) and Chain of Trust

The chain of trust is used to establish a trust relationship between the Certificate Authority (CA) and different entities. The CA generates a digital signature for the entity certificate to ensure integrity and authenticity. The leaf certificate of the chain, also known as the end-entity (EE) certificate, is associated with the specific entity being authenticated. During the verification process, the verifier has to build a candidate path from the EE certificate to a CA certificate in its trust store. According to the validation algorithm outlined in Section 6 of RFC5280, as a part of this chain building exercise, the verifier checks whether each certificate on the candidate chain has an *issuer* name matching the *subject* name of its predecessor, the presumed issuer. This is known as name chaining (ℕℂ), and corresponds to step ❷ in Section 1.

According to RFC5280 [2], both the *issuer* and *subject* name fields are values of the type Distinguished Name (`DN`). The type `DN` is defined as a sequence of Relative Distinguished Name (`RDN`), where `RDN` is a set of attribute type and value pairs. An example attribute type in `RDN` is the Common

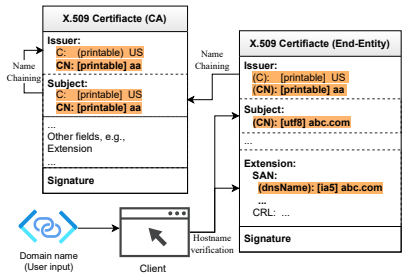

Fig. 1. Verification tasks, fields and extensions related to names

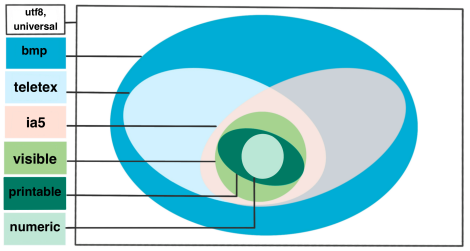

Fig. 2. Hierarchy of the charsets of common string types used to encode DN and SAN attributes

Table 1. Name attributes and their string types

Field	Attribute Type	Attribute Value String Type and Length Constraint (if any)
Issuer / Subject (type DN)	Country (C)	*printableString*/ *numericString* SIZE (2) / SIZE (3)
	Organization	*directoryString* SIZE (1 - 64)
	Organizational unit	*directoryString* SIZE (1 - 32)
	Distinguished name qualifier	*printableString*
	State/ Province name	*directoryString* SIZE (1 - 128)
	Common name (CN)	*directoryString* SIZE (1 - 64)
	Serial number	*printableString* SIZE (1 - 64)
	Locality	*directoryString* SIZE (1 - 128)
SAN	dnsName (dns)	*ia5String* SIZE (1-128)
	rfc822Name	*ia5String* SIZE (1-255)
	iPAddress	*OCTET* (8 octets)

Table 2. X.509 string types and their charsets

String Types	Description, encoding, and charset
bmpString	Basic Multilingual Plane of ISO 10646, 2-byte fixed length encoding for each char
ia5String	The charset is synonymous with ASCII (International Alphabet 5)
numericString	Digits (0–9) and SPACE
printableString	Character set: a–z, A–Z, 0–9, '()+-./:=? and SPACE
teletexString	CCITT/ITU-T T.61 character set, 1-byte fixed length encoding for each char
universalString	ISO 10646 charset in UTF-32 encoding (four-byte fixed length for each char)
utf8String	ISO 10646 charset in UTF-8 encoding (variable length from 1 to 4 bytes for each char)
visibleString	ASCII characters without control codes
directoryString	A general term, choose from { *utf8String*, *bmpString* *teletexString*, *printableString*, *universalString* }

Name (CN). A detailed list of attribute types, as well as the string types and length constraints of their attribute values, can be found in the Issuer/Subject rows of Table 1.

The rules for matching a CA's subject (DN1) and an entity's issuer (DN2) defined as follows [2]: 1) DN1 and DN2 have the same number of RDNs, 2) for each RDN in DN1, there is a matching RDN in DN2 in the same attribute, and 3) for each matching RDN pair, the RDNs' type should be the same and the values should match exactly after running the string preparation (stringprep) algorithm. For example, if the CA's subject DN has *[CN]=<utf8String>'aa'* and the EE's issuer DN has *[CN]=<ia5String>'aa'*, the two should not be considered a match because of a *type mismatch* . Moreover, since the *ia5String* type is not allowed in the CN attribute (see Table 1), one can also argue that the EE's issuer DN has a syntax error. In any case, this certificate chain should fail in the $\mathbb{NC}$ task.

Additionally, the string value of an attribute should be made of legal characters with respect to the charset of its string type. Otherwise, one can again argue for a syntax error on the certificate. For instance, a certificate with *[CN]=<printableString>'aa!'(0x616121)* should be invalid, as '!' is not legal within the charset of the *printableString* type.

As discussed above, issuer and subject attribute strings are processed by stringprep before matching. X.509 implementations *"MUST use LDAP stringprep profile, as specified in RFC 4518 [37], as the basis for comparison of distinguished name attributes encoded in either printableString or utf8String"* (Section 7.1 of RFC5280 [2]). The LDAP stringprep imports detailed processing rules from RFC3454 [20], specifying the standard character handling rules for Unicode strings. In short, it lifts a string to Unicode by mapping characters to Unicode code points, applies Unicode normalization, then checks for illegal characters and removes insignificant characters such as redundant space characters. Following a successful stringprep, implementations should be able to match a CA's

subject DN *[CN]= 'AA\x00\x08' (0x41410008)* to EE's issuer DN *[CN]= 'aa'*. Interestingly, as we will demonstrate later, the LDAP stringprep is often not followed by open-source implementations.

2.3 Hostname Verification ($\mathbb{HV}$)

After establishing the chain validity, hostname verification ($\mathbb{HV}$) is performed to ensure the EE's identity aligns with the application's input. Incorrect implementations of hostname verification can potentially lead to name confusion, which might in turn open doors to impersonation attacks.

Although $\mathbb{HV}$ is also essentially a string matching task, its rules differ from that of $\mathbb{NC}$. Hostname definition can trace back to RFC1123 [3]. It says the hostname can only include [a-zA-Z0-9], dash ('-'), and dot ('.'). In X.509, the dnsName attribute in SAN field of extensions is used to store domain-based hostnames. For hostnames represented by the dnsName attribute, implementations should adhere to the descriptions in [2, 23, 28, 29]. In short, the verifier implementation should check for disallowed characters, and then apply case-insensitive matching while honoring wildcards. Because dnsName can only be of the *ia5String* type (see Table 1), in order to fit in its charset, internationalized domain names (IDNs) need to be converted to Punycode [12] before getting stored in SAN at certificate issuance time. During verification, the implementation should apply the STD3AsciiRules (STD3) to further restrict characters from *ia5String*, as some ASCII characters, such as control codes, cannot be part of a valid domain name.

To honor wildcards, there are several constraints: 1) a wildcard should only appear and affect the leftmost subdomain, 2) it should not be used in second-level domain and top-level domain, and 3) a wildcard on the right hand side of character(s) is not allowed. For instance, *[dns]='*.a.a'* is acceptable, but *[dns]='a.*.a'* and *[dns]='x*.a.a'* are not; and *[dns]='*.a.a'* can match *[in]='a.a.a'* but not *[in]='b.a.a.a'*.

In terms of $\mathbb{HV}$, a point of contention is the role of EE's CN attribute in the subject field. According to some specifications [2, 30], if the dnsName attribute is not present on the EE certificate, the EE's CN attribute can be checked for $\mathbb{HV}$. When the CN attribute is used in $\mathbb{HV}$, stringprep, as described in Section 2.2, should still apply, and the restrictions on string type and charset outlined in Tables 1 and 2 also apply. Additionally, there is no explicit wildcard matching rules for the CN attribute.

In the rest of the paper, $\mathbb{HV}$-SAN refers to the case of $\mathbb{HV}$ performed on SAN, and $\mathbb{HV}$-CN refers to the case of $\mathbb{HV}$ performed using CN when a SAN with dnsName is absent on the EE certificate.

3 Problem Definition and Motivations

Here we discuss the name matching problem and motivation behind our methodology.

Problem definition. In this paper, the main question we ask is whether for a given implementation of X.509 validation, the semantics of name matching are equivalent to the one prescribed in the standard. Simply put, we ask the question: *in the context of $\mathbb{NC}$ and $\mathbb{HV}$, are the instantiations of the abstract* NameMatching *function in implementations correctly following the standards*?

Abstractly, the function NameMatching takes two names, $\texttt{Name}_1$ and $\texttt{Name}_2$, and returns a Boolean signifying whether they match or not. Logically, NameMatching can be further broken down into 3 functions: CheckTypes, Transform, MatchStrings. First, CheckTypes checks whether $\texttt{Name}_1$ and $\texttt{Name}_2$ have correct string types for their attributes (Table 1), and whether all characters in an attribute are legal with respect to its string type (Table 2). Then, Transform performs the canonicalization of $\texttt{Name}_1$ and $\texttt{Name}_2$ and returns $\texttt{CanonicalName}_1$ and $\texttt{CanonicalName}_2$. As an example, Transform should be instantiated with the logic of LDAP stringprep for $\mathbb{NC}$. Finally, MatchStrings checks whether attributes in $\texttt{CanonicalName}_1$ and $\texttt{CanonicalName}_2$ have matching types and string values. Depending on the attribute string types, MatchStrings might use different matching logic (*e.g.*, exact bitwise, case-insensitive, wild card matching, *etc.*).

Our investigation mainly focuses on identifying semantic bugs that makes an implementation of `NameMatching` to deviate from the standards. Although not the main focus, discovering memory safety bugs can be a by-product of this investigation. Bugs in any of the 3 functions that constitute `NameMatching` can make it return a wrong verdict on whether $\texttt{Name}_1$ and $\texttt{Name}_2$ match. For instance, a loose `CheckTypes` might allow certificates with names made of illegal characters to be accepted. Likewise, improper canonicalization in `Transform` and wrong matching logic in `MatchStrings` can incorrectly recognize two strings as a match or mismatch. These bugs are undesirable in $\mathbb{NC}$, because they can lead to a loss of service (*i.e.*, when a trusted CA does not get recognized as the legitimate issuer), or a loss of performance due to extra signature verifications being performed (*i.e.*, when a trusted CA gets erroneously identified as a candidate issuer). In contrast, bugs in $\mathbb{HV}$ can have more severe security repercussions, potentially opening doors to impersonation attacks.

Motivation for name field-specific testing. A design choice for testing is to decide whether to test the implementation end-to-end or to just focus on the name checking fields. In this discussion, we consider *end-to-end testing* to be the approach where one crafts input certificate chains that simultaneously test the general enforcement of semantic requirements on many different certificate fields and extensions. Examples of such an approach include FrankenCert [4], Mucert [11], Symcerts [7], and RFCCert [10]. In contrast, examples of field-specific testing include HVLearn [32], Morpheus [36], and an earlier work by Chau *et al.* [8]. In our investigation, we follow the field-specific approach and focus only on name-related fields. This choice is motivated by the following reasons. First, the potential search space for finding incorrectness in an X.509 library is very large, and as a result, targeting specific fields is more likely to make the testing more focused and be able to exercise the subtle logic of name processing, essentially trading breadth for depth in the investigation. Second, when multiple fields of different semantics are simultaneously tested, the test results can become less interpretable. This is due to the fact that implementations tend to reject a certificate as soon as possible, typically when the first problem on a certificate is encountered. Consequently, if an input certificate chain with multiple problems in different fields gets rejected by an implementation, it is not immediately clear which problematic field triggered the rejection. Focusing on specific fields can improve the interpretability and explainability of the test results.

Motivation for a portfolio of testing approaches. Now we discuss the next design choice, which is picking testing approaches that are effective for our problem. Tried-and-true choices include *crafting inputs with domain knowledge*, *adaptive fuzz testing*, and *dynamic symbolic execution*, each of which come with their own advantages and disadvantages. Crafting meaningful test inputs requires domain knowledge, and can be challenging to scale up. Fuzz testing can increase the scalability, but the automatically generated test cases might not lead to easily interpretable results, and it might occasionally get stuck due to specific quirks of implementations. Symbolic execution provides a useful formula-based abstraction for finding logical bugs and might achieve good code coverage, but it can suffer from the problem of path explosion if the test subject makes a large number of branching decisions, and not all path constraints are easy to solve. In particular, we observed that certain X.509 implementations instantiate `MatchStrings` as $\texttt{Hash}(\texttt{CanonicalName}_1) \stackrel{?}{=} \texttt{Hash}(\texttt{CanonicalName}_2)$. Symbolically executing such program code and finding satisfiable input values is tantamount to attacking collision resistance of cryptographic hash functions, and the resulting path constraints are extremely difficult (if not impossible) to solve. Finally, a symbolic execution engine can typically only handle test subjects in an intermediate representation that it supports well, which often restricts the source language that it can handle.

All of the three testing approaches have relative strengths and weaknesses. Since our goal is to thoroughly investigate X.509 name implementations, we take a portfolio of approaches in which,

when feasible, we test each implementation with all of the applicable approaches. As we will show later, this allows our investigation to achieve a reasonably good combined coverage.

4 Experiment Setup and Preliminary Study

4.1 Test Subject Collection

From previous works and online open-source repositories, we collected common cryptographic, IPSec, and TLS libraries that may contain implementations of X.509. Then, we inspect their API documentation and source code if necessary, to determine if they have a usable implementation of $\mathbb{NC}$ and $\mathbb{HV}$. Libraries that do not have these are excluded from our analysis. In the end, we are left with 23 open-source X.509 libraries written in 9 different programming languages. The list of test subjects and their versions can be found in the anonymous repository listed in Section 10. For each of these libraries, we consult its API documentation and create test harnesses for $\mathbb{NC}$ and $\mathbb{HV}$.

Some libraries (*e.g.*, **WPA**) can outsource X.509 functionalities to other libraries (*e.g.*, **OpenSSL**). However, so long as they have their own internal implementation of X.509, we expose their internal implementations to our test harness to increase the diversity of our test subjects. We also note that some libraries, including the Apache ones, rely on Java's JSSE for chain validation (which includes $\mathbb{NC}$), but they implement and perform their own $\mathbb{HV}$.

4.2 Preliminary Study Using Inputs Crafted with Domain Knowledge

We first prepared a collection of test cases using domain knowledge. This approach is not meant to be an in-depth investigation on its own, nevertheless, as a preliminary study, it can help us determine problems that can be revealed without having to go deep into the execution paths. Specifically, the focus here is mainly on the *attribute lengths* and *charsets*. For example, we are interested to see whether certificates with zero-length or absent RDNs are accepted, whether constraints on string types and lengths are enforced, whether certain attribute types are supported and checked by an implementation, and whether implementations reject illegal characters with respect to the charset of the attribute string types. These questions can often be answered in the negative with one counterexample.

We created test cases based on the name and string specifications in references [20, 23, 31, 37], for which a summary can be found in Table 1 and Table 2. As shown in the tables, if present in an X.509 certificate, each attribute has a minimum and maximum length, as well as the acceptable string types, which in turn limits the permissible character set of the attribute. Our tests mainly focus on the `CN` and `SAN` fields of the name structure shown in Table 1, so all of our test cases leave the other certificate fields and extensions well-formed and valid. We ensure valid signatures by signing the test certificates with a self-generated private key. In the end, 2299 test inputs (certificate chains) were created for $\mathbb{NC}$, and 31 for $\mathbb{HV}$. We note that the NC test inputs were created by a simple script that enumerated straightforward combinations of *<attribute type, correct/incorrect string types>*, and *<string type, legal/illegal characters>*. The name attributes and string types, as well as what count as legal and illegal characters, correspond to Table 1 and Table 2. Because the charsets and string types are well-defined, it only took several man-hours of manual effort to inspect and identify problems in the test results.

4.3 Findings

Table 3 summarizes the findings from the preliminary study. There are indeed many violations regarding lengths of attributes and strings, as well as illegal string types and characters. For instance, in $\mathbb{NC}$, 18 libraries allow certificates with zero-length name attributes (*e.g.*, *[CN]=(None)*, marked $\#M_1$ in Table 3), which are syntactically malformed as they directly violate the string

Table 3. Findings from the preliminary study

Issues \ Libraries		axTLS	BearSSL	BoringSSL	Botan	GnuTLS	MatrixSSL	mbedTLS	OpenSSL	WolfSSL	StrongSwan	WPA	webpki	cert validator	Py-SSL	Go-Crypto	Erlang-OTP	BouncyCastle	SunJSSE	httpclient	CXF	PHP-SecLib	Node.js	Forge
ℕℂ	$\#M_1$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓			✓		✓
	$\#M_2$	✓									✓	✓												
	$\#E$	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	-	✓	✓	✓	✓	-		✓	-	✓
	$\#T$	✓		✓	✓			✓	✓		✓	✓					✓	✓	✓					✓
	$\#S$		✓			✓	✓			✓			✓			✓						✓		
𝕀𝕍	$\#W$	-	✓	✓	✓	✓			✓	✓	-			-		✓	✓	✓	✓	✓	✓	✓	✓	-
	$\#W_CN$		✓				✓	✓		✓					✓	✓		✓	✓	✓	✓	✓		
	$\#UC_1$		✓	✓	✓	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	
	$\#S$																					✓		

✓: detected -: not applicable M_i: missing attributes E: empty DN T: string Type mismatch
S: case Sensitive W: wildcard position issue W_{CN}: applying wildcard to CN in HV UC_i: unexpected char

lengths requirements in Table 1. Similarly, 16 implementations accept certificate chains with empty issuer/subject fields. According to the RFC5280 [2], *the issuer field MUST contain a non-empty* DN *value* (Section 4.1.2.4). Likewise, for CA certificates, *the subject field MUST be populated with a non-empty* DN *value* (Section 4.1.2.6). Thus, our test cases with the EE certificate's issuer field and the CA certificate's subject field set to an empty sequence, can be seen as malformed inputs that should not be accepted. 3 implementations accept a pair of issuer and subject names even when they do not have the same number of attributes ($\#M_2$ in Table 3). This deviates from the standard's DN matching rules outlined in Section 2.2. For example, we found that **axTLS** ignores some RDN attributes, such as `Surname`, when it tries to match DNs.

Notably, in ℕℂ, we found 12 implementations accept mismatched attribute string types (Findings $\#T$ in Table 3). For example, the types *utf8String* and *printableString* are matched by **OpenSSL**. Because of this, some Unicode characters outside the charset of *printableString*, can also be put in an attribute of the type *printableString*, which **OpenSSL** and other libraries would accept. In other words, the attribute types and their corresponding charsets are often neglected by implementations, many of which are willing to accept malformed inputs. In 𝕀𝕍, we found that illegal characters outside the `STD3` ASCII range, can be accepted by all implementations ($\#UC_1$ in Table 3). For both ℕℂ and 𝕀𝕍, some libraries incorrectly use case-sensitive matching in their instantiations of `MatchStrings` ($\#S$ in Table 3). Furthermore, some libraries incorrectly perform wildcard matching in 𝕀𝕍-CN and when wildcard appears in wrong positions ($\#W_{CN}$ and $\#W$ in Table 3).

5 Automated Testing

To further investigate other aspects of name processing, including the support for stringprep and wildcards, as well as intricacies in the string matching logic, we further apply 2 automated testing approaches. For the sake of scalability, in both testing approaches, we only consider a certificate chain with a length of 2. Certificate fields and extensions other than `CN` and `SAN` are left well-formed and valid. An overview of the high-level workflows can be found in Figure 3.

5.1 Adaptive Fuzz Testing

We developed a tool for performing adaptive black-box fuzzing to test the name-matching functions with automatically generated inputs. We chose the black-box setting so that this can be applied to all of our test subjects written in different programming languages. Several issues already detected in preliminary study can now be purposefully ignored during fuzzing, enabling us to focus on other aspects (*e.g.*, stringprep). The test harnesses created for preliminary testing are reused here.

Conceptually, our fuzzing tool generates concrete values for $\texttt{Name}_1$ and $\texttt{Name}_2$, which are then used to test an implementation of `NameMatching`. It applies mutations to certificate name fields, starting from a fixed set of well-formed seed certificates. A high-level description of our fuzzer's behavior is presented in Algorithm 1. For generating mutated certificates, we maintain an internal representation of a certificate, and mutate the chosen certificate fields. The mutated internal representation is then serialized into ASN.1 DER representation using an ASN.1 library [17]. This has to be done at the ASN.1 level but not the X.509 level, because regular X.509 libraries would forbid our mutated certificates due to their inbuilt syntactic checks. To further reduce the search space, we limit our tool to only modify name attributes of the EE certificate.

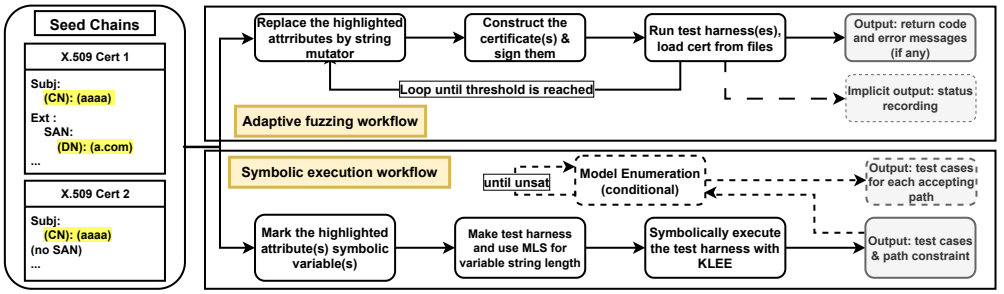

Fig. 3. Workflows of the automated testing approaches.

Mutation strategies. Our tool, outlined in Algorithm 1, uses diverse mutation strategies. Each strategy combines a string operator (add, replace, pad) with a character category, based on RFC 3454 [20] (e.g., control codes, spaces), refined by the ASCII boundary. A `get_char` function randomly selects characters within each character category. The `mutate` function calls `get_char` and applies a chosen mutation strategy to a string (S) at all positions, creating mutated strings (S'). These are then applied to a certificate (C), generating a set of test certificates (C_{ctx}). We avoid redundant test cases by recording and skipping previously generated ones. Accepted test certificates are stored and serve as seeds for further mutations, iterating until a *threshold*. The initial seeds included crafted certificates (Section 4) accepted by some libraries and those from symbolic execution. In the end, the fuzz testing generated 78919 test cases for $\mathbb{NC}$, 62877 for $\mathbb{HV}$-SAN, and 22518 for $\mathbb{HV}$-CN, shared across all test subjects.

Result inspection. Using our domain knowledge, we partitioned the input space into classes of inputs that should be accepted or rejected, with all inputs within a particular class sharing the same expected outcome. Input classes that abide by RFCs' matching standard (e.g., `caseInsensitiveMatch`) should be accepted, and others (e.g., inputs with multiple wildcards) should be rejected. After testing, a human-in-the-loop then looks at test inputs that got *accepted* by at least one test subject, and then based on the corresponding input classes, decides (1) if the acceptance is erroneous, and (2) whether the test subjects that rejected the inputs (if any) are erroneous. Altogether the manual effort needed was less than 8 man-hours.

5.2 Symbolic Execution

Since our investigation targets open-source implementations, instead of a symbolic execution engine that targets binary, we choose to use KLEE [5], which interprets LLVM bitcode generated from source code. Because not every programming language are easily convertible to the subset of LLVM IR that KLEE supports, we only use KLEE to test implementations that are written in C. For

each of the C libraries, we prepare one test harness for each of the $\mathbb{NC}$ and $\mathbb{HV}$ tasks. We override the signature verification function of our test subjects, similar to what previous work [7] did. Due to the semantics of the ASN.1 DER encoding, we avoid introducing symbolic ASN.1 lengths by following the strategies proposed by previous work [7], which only marks name *attribute value* bytes as symbolic. This way, all the lengths in ASN.1 DER take concrete values, which helps to get through the parsing code of the test subject and explore its string normalization and matching logic. The name attributes that are supported by a test subject (and are thus suitable for marking as symbolic) were determined by our preliminary testing effort.

To increase the discovery of interesting input values, we further incorporate the meta-level search (MLS) strategy [8] in our test harnesses, and apply the idea of model enumeration with blocking clauses on KLEE's output path constraints.

KLEE with meta-level search (KMLS). This strategy takes advantage of KLEE's path-forking behavior to automatically explore different combinations of input lengths with only one test harness [8]. The key idea is that the ASN.1 DER encoding of a certificate exhibits linear length constraints (*e.g.*, the length of a parent node is given by the summation of the length of its child nodes), which can also be symbolically executed and solved by KLEE. One can thus ask KLEE to determine satisfying assignments to the lengths of different certificate components in the test harness, and then programmatically pack the concrete and symbolic values into test certificates *before* symbolically executing the actual entry point function of the test subject.

We adapted this strategy in our test harnesses to enable the target name attributes in the certificates to have a short but variable length, which explores a larger search space without additional manual effort. This, however, could increases the run-time burden of KLEE and worsen the potential problem of path explosion. For the sake of practicality, we only allow the variable lengths to range from 1 to 5. Based on our results, this limit works well for most amenable implementations (except for **OpenSSL**).

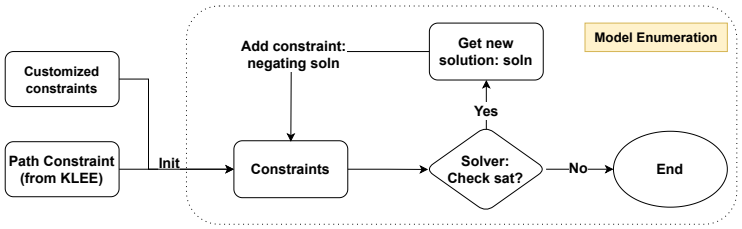

Fig. 4. Workflow of the model enumeration loop.

Model enumeration with blocking clause introduction. We perform model enumeration externally to KLEE, in an effort to enhance the test case generation. The key observation here is that given an extracted path constraint which abstracts the branching decisions taken by a particular execution path, the SMT solver does not always generate a test input that is interesting to us in the context of name checking, especially when the test subject implements a lax or incorrect checking logic to begin with. For example, if an implementation does not check the `Country` (C) attribute, the SMT solver might report a test input where the CA's subject `DN` with *[C]='AA'* is a match to EE's issuer `DN` with *[C]='AA'*. However, a more interesting test input that still satisfies the same path constraint would be one that shows CA's subject `DN` with *[C]='BB'* is a match to EE's issuer `DN` with *[C]='AA'*. This can be achieved by iteratively querying the SMT solver, and refining the path constraint in each iteration by adding a new conjunction with the negation of the satisfying

assignment found in the previous iteration (a blocking clause), as depicted in Fig. 4. This iteratively narrows down the search space and pushes the SMT solver to discover more satisfying assignments that might be of interests.

Nevertheless, the model enumeration loop can sometimes discover many similar test inputs that are uninteresting. For instance, if a name attribute has the wildcard character, the loop will end up discovering numerous satisfiable assignments that can trivially match the wildcard. To further reduce the search space and discover more potentially interesting test cases, we introduce additional clauses (referred to as *customized constraints* in Fig. 4) to the path constraint from KLEE, before the model enumeration loop begins. These clauses include constraints that prohibit wildcards and restrict the charset (*e.g.*, the STD3 rule) in the satisfiable assignments.

5.3 Findings

In the following subsections, we present our findings on the $\mathbb{NC}$ and $\mathbb{HV}$ tasks respectively. For each individual finding, the automated testing approaches that were able to discover it are also marked in Table 4. As discussed in Section 3, some implementations perform string matching on hashed names, which cripples symbolic execution. This is why KMLS was not applicable to $\mathbb{NC}$ in **BearSSL**, **GnuTLS**, **MatrixSSL**, and **WolfSSL**, but has much more success with them in $\mathbb{HV}$. We note that the automated testing approaches can also reproduce findings like $\#S$, $\#T$, and $\#UC_1$ in Table 3. For simplicity, Table 4 only presents the new findings uncovered by automated testing.

5.3.1 Findings on $\mathbb{NC}$. **Insignificant characters remain significant.** As discussed in Section 2.2, implementations should apply the stringprep algorithm to preprocess name strings, a step of which is to map insignificant control characters to nothing (effectively removing them) prior to matching. However, we found that all of the implementations failed to faithfully follow this requirement. For instance, none of the libraries accepted the certificate chain when the EE certificate's issuer DN has *[CN]= 'aa'* and the CA certificate's subject DN has *[CN]= 'aa\u06dd'*. The two should match, because, according to stringprep, \u06dd should be mapped to nothing. Interestingly, we observed that some libraries are partially correct with respect to stringprep. For example, both (and only) axTLS and SUN can accept *[CN]=aa\0\0* as a match of *[CN]=aa*. Additionally, only **Botan** and **MatrixSSL** can match strings with leading \r and trailing \0, but they also missed the other insignificant characters. In any case, the big picture is that stringprep's mapping rules are not faithfully followed by implementations.

Name confusion. The issue of name confusion arises from the fact that the verification process can incorrectly match visually and logically distinct strings. This issue is observed in only two libraries, **axTLS** and **StrongSwan**. For **axTLS**, it mishandles truncation of string attributes on certificates. Specifically, **axTLS** treats 'aa\0\0a' as a match of 'aa'. To understand this phenomenon, we performed root cause analysis, and found that the reason is because **axTLS** uses some C string functions which treats the first occurrence of the \0 character as the null terminator. This means that even the partial handling of *insignificant character* (mapping to nothing) discussed previously is likely a mere coincidence, not that **axTLS** actually implements a subset of stringprep. For **StrongSwan**, through root cause analysis, we found that whenever the certificates already have SKI and AKI, it uses those for building a chain (instead of performing the traditional $\mathbb{NC}$ as stipulated in the standard [2]). This also explains why **StrongSwan** has $\#M_1$, $\#M_2$, and $\#E$. We will discuss the implication of this way of chain building in Section 7.

Name confusion caused by ignoring sting types. We found that **OpenSSL** matches *[CN]= <teletexString> '$'(0xa4)* to *[CN]='¤'(0xc2a4)* (Unicode code point a4). In this example, the code point a4 is $ in T.61 (*teletexString*) but ¤ in Unicode. Here, the implementation only tries to match the code

Table 4. New findings from automated testing

Name	Ver.	Task	Findings	KMLS	Fuzz
axTLS	2.1.5	$\mathbb{NC}$	$\#N$	✓	✓
BearSSL	0.6	$\mathbb{HV}$	$\#UC_2$	✓	✓
			$\#N$	✓	✘
BoringSSL	chromium -stable	$\mathbb{HV}$	$\#UC_2$	-	✓
OpenSSL	1.1.1o 3.1.0	$\mathbb{NC}$	$\#N$	✓	✘
		$\mathbb{HV}$	$\#N$	✓	✘
			$\#UC_2$	✓	✓
OpenSSL	1.1.1o 3.1.0	$\mathbb{HV}$	$\#UC_3$	✓	✓
WolfSSL	5.1.1	$\mathbb{HV}$	$\#OR$	✓	✘
StrongSwan	5.9.7	$\mathbb{NC}$	$\#N$	✓	✓
Erlang-OTP	22	$\mathbb{HV}$	$\#N$	-	✓
phpseclib	3.0.11	$\mathbb{HV}$	$\#RE$	-	✓
Node	17.9 (open)	$\mathbb{HV}$	$\#UC_2$	-	✓
			$\#UC_3$	-	✓

✓: detected ✘: not detected -: not applicable

RE: unexpected regex match *N*: name confusion UC_i: unexpected char *OR*: out-of-bound read

point of the encoded bytes, while assuming all string types follow the Unicode mapping between charset and code points. In other words, the implementation lacks support and differentiation for the T.61 charset, but does not reject certificates that use such string type. This issue is marked as $\#N$ in Table 4.

5.3.2 Findings on $\mathbb{HV}$. Because the $\mathbb{HV}$ can potentially match either CN or SAN, and the two have different attribute definitions (see Section 2.3), here we present our findings with the suffix -CN or -SAN to indicate the applicable field. We omit the suffix if a finding applies to both CN and SAN.

Discrepancies in using CN for $\mathbb{HV}$-SAN. As discussed in Section 2.3, whether CN should be used in $\mathbb{HV}$ is a point of contention and differs depending on the target application standards. Unsurprisingly, there are discrepancy related to this. We found that when SAN exists on the certificate, only the `checkHost` functions in **`Node.js`** before version 18 will by default *also* consider CN as a hostname candidate. Among most libraries, CN is used as a last resort, only when SAN is absent.

Illegal hostnames allowed in $\mathbb{HV}$-SAN. Through our experiments, we found that many libraries accept illegal hostnames in the `dnsName` attribute of certificates. As discussed in Section 2.3, although the `dnsName` attribute technically has a string type of *ia5String*, the general expectation is that it should be made of a more restrictive set of characters (STD3), and there are also some restrictions on where the wildcard character can appear. Interestingly, most libraries do not strictly enforce this, and have leniency in parsing and processing the `dnsName` attributes that contain illegal characters (findings $\#UC_1$ in Table 4). We also found that these illegal characters (*e.g.*, `!`) can be matched against a wildcard character. `dnsName` with illegal characters can be seen as syntactically invalid inputs, and the expectation is to reject them, possibly during parsing.

A related but arguably more severe issue, which allows two seemingly different domains to match, exists in both **`BearSSL`** and **`OpenSSL`**. For instance, **`OpenSSL`** can accept code points beyond ASCII at the starting position of a `dnsName`, so long as they are adjacent to the character `.` (dot). Consequently, for *[in]='.a'* and *[dns]=<ia5String> (0xF1A2BEBE2E61)* (*i.e.*, the code point is ñ¢¾¼.a in UTF-8), these two are considered to match, even though the extra prefix code points in `dnsName` renders it an illegal hostname. This bug also affects **`Node.js`** if it is configured to use **`OpenSSL`**, as well as previous versions of **`BoringSSL`** (findings $\#UC_2$ in Table 4).

Additionally, **`OpenSSL`** can also match *[CN]='A.a'* with *[in]='.a'*. In fact, the character *A* can even be replaced by [a-zA-Z0-9] and '$-[],+^', etc. Note that some of these characters render CN to contain illegal hostnames, and we marked this as $\#UC_3$ in Table 4. We investigated the **`OpenSSL`** source

code and found that its implementation treats a leading dot as a cue to perform suffix matching, and thus the left-most subdomain gets ignored.

Wildcard issues in $\mathbb{HV}$-SAN. A special case to the aforementioned illegal hostname problem concerns the processing of the wildcard character. We have identified two types of issues related to this. The first issue concerns the lack of restrictions on where the wildcard character can appear in a `dnsName` attribute. For instance, **`WolfSSL`**, **`Apache-CXF`**, and others do not limit the wildcard's position, allowing it to appear at the top-level domain. Similarly, **`Go-Crypto`**, **`Apache-HttpClient`**, and some others allow wildcard to appear at the second-level domain. Wildcards at the non-leftmost position are also accepted by many libraries. Some libraries, such as **`OpenSSL`**, even accept multiple wildcards in a `dnsName` attribute. All of these deviate from the standard requirements discussed in Section 2.3. We note that a subset of these findings were also discussed in a previous research [32], and it appears that this issue has persisted in several libraries for many years. The second issue is that wildcard matching can be performed incorrectly sometimes. For instance, we found that *[dns]='a*b*'* can match *[in]='aaaa'* in **`WolfSSL`**. Both of these issues are marked as #*W* in Table 4.

Name confusion in $\mathbb{HV}$-CN. We found a name confusion bug in **`Erlang-OTP`**, which ignores the heading '.' (dot) in a string during $\mathbb{HV}$-CN. For instance, both *[CN]='..a'* and *[CN]='.a'* can be incorrectly matched with *[in]='a'*. Additionally, we also found name confusion in the implementations of $\mathbb{HV}$-CN in **`OpenSSL`** and **`BearSSL`**. This is basically the reincarnation of the T.61 handling problem discussed in Section 5.3.1. Basically, both **`OpenSSL`** and **`BearSSL`** can parse `CN` of the *teletexString* type, but always assumes the Unicode mapping of code points. Because of this, both of them can match *[CN]=<teletexString>'$'(0xa4)* to *[in]='¤'(0xc2a4)*. All these are marked as #*N* in Table 4.

Wildcard misuse in $\mathbb{HV}$-CN. We also found that 11 implementations apply wildcard matching rule when performing $\mathbb{HV}$-CN (findings $\#W_{cn}$ in Table 4). For instance, some implementations can match *[CN]='*.a'* with *[in]='a.a'*, which deviates from the standard (see Section 2.3).

Regex misuse in $\mathbb{HV}$. We found that **`PHP-SecLib`** misused regex match in its implementation of the $\mathbb{HV}$ task, causing some special characters to function in the regex semantics (#*RE* in Table 4). As an example, for $\mathbb{HV}$-CN, **`PHP-SecLib`**'s `validateURL` function thinks *[CN]='a+'* is a match for *[in]='aa'*. Furthermore, *[CN]='[\D.]+'* can match any domain names without numeric characters, *e.g.*, *[in]='google.com'*. For $\mathbb{HV}$-SAN, these two examples also work (using the `dnsName` attribute).

Out of bound read in $\mathbb{HV}$. We found that **`WolfSSL`** has an out of bound read problem in its implementation of $\mathbb{HV}$. The issue is caused by an improper length check on the input string *[in]*. In short, if *[in]* is not null terminated, the program code could read beyond its buffer boundary, possibly until a null terminator is reached. This issue is marked as #*OR* in Table 4.

6 Coverage Evaluation

We empirically evaluate coverage achieved by our investigation. The goal of this evaluation is not about comparing the merit or efficiency of different approaches. The objective here is to show that collectively, our investigation achieved better coverage in the specific context of name processing. Our evaluation considers $\mathbb{NC}$, $\mathbb{HV}$-SAN, and $\mathbb{HV}$-CN.

Baselines, setups and metrics. We select the seminal work FrankenCert [4] and HVLearn [32] as representative baseline for $\mathbb{NC}$ and $\mathbb{HV}$ task, respectively. FrankenCert performs *end-to-end* black-box fuzzing, as it mutates many different certificate fields and extensions at the same time. On the other hand, HVLearn by design only explores $\mathbb{HV}$, so it is not used as a baseline for $\mathbb{NC}$. We also considered other recent works including MuCert [11] and SADT [27], but because their source code files are unavailable online, we resort to using FrankenCert as the baseline for $\mathbb{NC}$. In our evaluation, we measure code coverage at the line, function, and branch level.

Algorithm 1 Fuzzing Algorithm

```
Input: C , S , M
Output: Store
Init: idx ← 0; N ← threshold; Store ← seed
while threshold do
    while N do
        C ← getCertToMutate(Store, idx)
        T ← getMutationStrategy(M, random())
        S ← getString(C); S′ ← mutate(S, T)
        C_ctx ← packageToAsn1(C, S′)
        for C′ in C_ctx do
            if testHarnessAccept(C′, S) then
                Store ← add(Store, C′)
            end if
        end for
        Update: N ← update(N)
    end while
    Update: idx ← update(idx);
    Update: threshold ← threshold − 1;
end while
```

Explanation of variables:
C for current certificate
S for string to be mutated
M for mutation strategies
N for the threshold of mutation
$Store$ for the store of accepted certificate chains

Table 5. Example of ℕℂ/𝕀𝕍 task's coverage

(a) Coverage statistics for ℕℂ

Library	Coverage	Approach					
		FrankenCert(2M)	FrankenCert(8M)	Craft	Fuzz	KMLS	C+F+KMLS
GnuTLS	Line	3211(4.2%)	3211(4.2%)	3407(4.5%)	3504(4.6%)		**3530(4.6%)**
	Func	267(6.3%)	267(6.3%)	278(6.6%)	**285(6.7%)**	N/A	**285(6.7%)**
	Branch	1643(3.0%)	1643(3.0%)	1728(3.1%)	1783(3.2%)		**1812(3.3%)**
OpenSSL	Line	6324(4.2%)	6324(4.2%)	5322(3.5%)	6419(4.2%)	6334(4.2%)	**6565(4.3%)**
	Func	887(8.2%)	887(8.2%)	796(7.3%)	902(8.3%)	892(8.3%)	**903(8.3%)**
	Branch	2745(3.0%)	2745(3.0%)	2291(2.5%)	2755(3.0%)	2711(3.0%)	**2841(3.1%)**

(b) Coverage statistics for 𝕀𝕍

Library (Task)	Coverage	Approach				
		HVLearn	Craft	Fuzz	KMLS	C+F+KMLS
OpenSSL (𝕀𝕍-SAN)	Line	3703(2.4%)	3688(2.4%)	3710(2.5%)	3732(2.5%)	**3749(2.5%)**
	Func	592(5.5%)	593(5.5%)	593(5.5%)	593(5.5%)	**594(5.5%)**
	Branch	1540(1.7%)	1489(1.6%)	1530(1.7%)	1579(1.7%)	**1588(1.7%)**
OpenSSL (𝕀𝕍-CN)	Line	3680(2.4%)	3627(2.4%)	3641(2.4%)	3926(2.6%)	**3939(2.6%)**
	Func	587(5.4%)	585(5.4%)	584(5.4%)	**593(5.5%)**	**593(5.5%)**
	Branch	1524(1.7%)	1469(1.6%)	1479(1.6%)	1699(1.9%)	**1708(1.9%)**
GnuTLS (𝕀𝕍-SAN)	Line	1820(2.4%)	1799(2.4%)	1819(2.4%)	2115(2.8%)	**2116(2.8%)**
	Func	146(3.5%)	146(3.5%)	146(3.5%)	**162(3.8%)**	**162(3.8%)**
	Branch	974(1.7%)	947(1.7%)	970(1.7%)	1138(2.0%)	**1141(2.0%)**
GnuTLS (𝕀𝕍-CN)	Line	1937(2.5%)	1869(2.4%)	1952(2.6%)	2400(3.1%)	**2405(3.1%)**
	Func	151(3.6%)	145(3.4%)	151(3.6%)	**173(4.1%)**	**173(4.1%)**
	Branch	1030(1.9%)	995(1.8%)	1045(1.9%)	1303(2.3%)	**1306(2.3%)**

For FrankenCert, we gave it the same seed certificate chains that we used in our adaptive fuzzing. The number of extensions on all of the seed certificates is 4, which include the basic constraint, SAN, SKI, and AKI. Because of this, in FrankenCert's configuration file, we limit the number of extensions to 4. Despite this limit, FrankenCert is still free to randomly mutate values of any fields and extensions due to its end-to-end design. This larger search space gives Frankencert some potential advantages in achieving a higher coverage than our name-field specific testing. For fairness, we used FrankenCert to generate 8 million test cases, same as its original paper [4].

HVLearn attempts to learn a state machine that abstract the name matching logic by querying the test harnesses. Because of this, HVLearn generates a state machine model instead of a set of test cases when it finishes. To approximate its coverage, we print all the query strings that HVLearn issued to the test harness, and then package them into our seed certificate to obtain concrete certificate test cases. This enables a fair comparison and helps to avoid the potential influences of other certificate fields and extensions that are not related to 𝕀𝕍. The number of resulting test cases for each test subject varies, ranging from the minimum of 157 test cases (**mbedTLS**) to the maximum of 3863 test cases (**GnuTLS**).

Recall that, our portfolio of testing approaches consists of test cases crafted with domain knowledge (denoted as Craft or simply C), fuzz testing (denoted as Fuzz or just F), and KLEE with meta-level search (denoted as KMLS).

We note that KMLS was able to finish exploring ℕℂ and 𝕀𝕍 of different implementations with varying execution time. The only exception was the ℕℂ task of **OpenSSL**, for which we terminated after running for 30 days. Nevertheless, KMLS was able to explore a large number of execution paths in that period. A detailed breakdown of the KMLS statistics can be found in the anonymous repository in Section 10. Similar to what we did for HVLearn, for each test subject, we also package the satisfiable assignments found by KLEE into our seed certificates to obtain concrete test cases for coverage evaluation. For simplicity, we do not include the test cases found by model enumeration in this evaluation, so the measured coverage of KMLS is a lower bound of what our investigation has achieved.

Finally, all the concrete test cases were replayed to their corresponding applicable test subject, and we use `lcov` to collect their coverage statistics. We also note that because the portfolio of test approaches target slightly different search spaces (Section 4), this evaluation is not meant to be comparing the coverage of the 3 test approaches. The main point here is that they collectively yield a good coverage for our investigation.

Results. Due to page limit, we show **`OpenSSL`** and **`GnuTLS`** as two representative examples in Table 5. A more complete coverage result can be found in the anonymous repository listed in Section 10.

The first interesting observation is that the test cases crafted with domain knowledge can already yield a decent coverage. Second, our portfolio of approaches collectively delivers a better coverage than the 2 baselines. In fact, even when FrankenCert is given more leeway to randomly mutate other fields and extensions, our portfolio of approaches can often exercise more lines of code, more functions, and more branches in total, under a more limited focus on only `CN` and `SAN`. As shown in Table 5, Frankencert with 2 million and 8 million test cases achieved the same coverage. When compared to HVLearn, our portfolio also yields more code coverage. This is perhaps unsurprising, as the queries in HVLearn are all made of legal *ia5String* characters. In other words, mismatched attribute types and illegal characters are not considered in its exploration.

7 Discussion

7.1 Limitations and Future Work

In our study, we employed three distinct approaches for name testing. Despite this, for the sake of practicality, our tests do not achieve full completeness. We are also not claiming that the approaches and tools we used are necessarily the best in class. For instance, coverage-guided greybox fuzzing has been shown to be another promising bug-finding approach. It is conceivable that future work can achieve further improvements in terms of coverage, and potentially discover yet more bugs related to name checking. Nevertheless, the results of our investigation can be seen as a meaningful lower bound of name-related bugs in X.509 implementations. More importantly, we believe this work provides meaningful and ample data points to empirically demonstrate the engineering challenges surrounding names in X.509, highlighting the gaps between standards and implementations.

Another challenge we faced was the lack of a test oracle. Differential testing might not be very effective, as multiple implementations can all have the same deviation (*e.g.*, practically none of the tested libraries implement the required stringprep). We worked around this challenge by involving human-in-the-loop, which requires domain knowledge. Although a recent effort managed to derive a formally verified implementation of a subset of X.509 [16], we note that it does not contain a verified implementation of stringprep or $\mathbb{HV}$. Developing a programmatic oracle for testing implementations of $\mathbb{NC}$ and $\mathbb{HV}$ remains an open problem.

7.2 Severity and Pathways to Exploitation

Curious readers might wonder what are the severity and how to exploit the findings discussed in Sections 4.3 and 5.3. We note that several real-world operational factors can affect the success of an end-to-end exploit. For example, whether one is able to register for a company or organization with special characters, and whether CAs are willing to issue certificates containing special characters.

Take the regex match problem of **`PHP-SecLib`** as a representative example. As mentioned in Section 5.3, a certificate with *[CN]='[\D.]+'* can match any input hostnames (*e.g.*, *[in]='g.co'*). In order to obtain such a certificate, the attacker would first have to register an organization with the name [\D.]+. We note that the list of characters allowed in organization names varies in different jurisdictions. A quick investigation online suggests that this name might be permitted in the UK under *The Company, Limited Liability Partnership and Business (Names and Trading Disclosures)*

Regulations 2015 (Section 2 [21] and SCHEDULE 1 [22]). If an attacker successfully registers an organization, then the next step is to obtain a certificate from one of the trusted commercial CAs. Typical CAs usually can issue multiple types of certificates at different validation levels, including domain validation (DV), organization validation (OV), and extended validation (EV) certificates. Although CAs usually require proof of ownership of a domain name for DV and EV certificates, that is usually not necessary for OV certificates. In theory, with the legal documents of organization registration, the CA might be convinced into issuing an OV certificate with the attacker chosen name. Whether a CA forbids certain characters in the `CN` attribute depends upon its operational policies, and can vary from CA to CA.

Generally speaking, to exploit the other findings on $\mathbb{HV}$, the attacker would follow the same template outlined above. The success of such attacks often depends on whether special characters can slip through company registrations and CA's vetting of signing requests. Because registering organizations in many different jurisdictions and obtaining certificates of various validation levels from different CAs are both are costly and time-consuming, we refrain from actualizing end-to-end exploits. Nevertheless, relying solely on the issuing CAs to maintain compliance with standards might not be the best approach, as some of them can also fumble in their operations and issue non-compliant certificates [25]. We recommend implementations to tighten their verification and reject obvious cases of invalid inputs, regardless of the operational policies of CAs.

In contrast, the findings on $\mathbb{NC}$ tend to have smaller security impact than their $\mathbb{HV}$ counterparts, mainly due to the chain validity backed by digital signatures. However, their compatibility issues may lead to less critical but undesirable outcomes. For instance, an implementation that performs case sensitive matching (*e.g.*, **`WolfSSL`**) might fail to recognize a legitimate CA certificate, thus failing to build an otherwise valid chain of trust. On the other hand, an overly permissive implementation might discover incorrect candidates that do not form a meaningful chain of trust. Verifying the signatures of these candidate chains mitigate potential security problems, but could be a waste of computational resources and time.

7.3 Bug Reports and Feedback from Developers

We reported our findings to the corresponding maintainers of implementations, and received some interesting responses. First, the **`PHP-SecLib`** regex match problem has been acknowledged and fixed by the developers. One CVE has been assigned to that finding. Second, the out of bound read problem of **`WolfSSL`** has also been confirmed and fixed by its developers. Another CVE has been assigned to that. The wildcard matching problem of *[dns]='a*b*'* in **`WolfSSL`** was also confirmed and partially addressed. The developers mentioned that they want to support this as a feature, so instead of our recommendation of rejecting *[dns]='a*b*'* due to it being an invalid hostname, they opt to correct the matching logic instead. The name confusion problem due to heading dot in **`Erlang-OTP`** has been confirmed ans fixed as well.

The development team at **`OpenSSL`** confirmed the problem of $\#UC_2$ in $\mathbb{HV}$-SAN, and indicated that plans are being made for future fixes. Interestingly, upon receiving our report of the *teletexString* string problem ($\#N$), they agreed that the current way of handing non-Unicode code points is not ideal, but considered this a non-security-critical issue. Similarly, the author of **`BearSSL`** attributed the bugs primarily to unclear definitions, and noted that embedding large mapping tables for stringprep (see Section 2.2) consumes significant amount of memory, which is not ideal for resource-limited devices. The developers at **`StrongSwan`** confirmed that when the name chaining process fails, its implementation only prints a warning text and does not result in a verification error. They instead rely on SKI and AKI for chain building. Discussions with **`SunJSSE`**, **`OpenSSL`**, **`GnuTLS`**, and **`Botan`** are still ongoing. Some developers are not as active, and we are still waiting for their responses.

We wish to express our gratitude to the developers for their timely responses and valuable feedback. The objective of this study is not to criticize the developers but to underscore the difficulties involved in implementing a faithful X.509 validation according to the complex standards, and open dialogues on whether the community really benefits from such standards. The feedback from developers of **OpenSSL** and **BearSSL** actually echoes with our stance that the current specification is unnecessarily complex, and a standard that no one follows is perhaps not a good standard.

7.4 Rethinking the Merits and Necessity of Complex Names

For $\mathbb{NC}$, although the certificate path validation algorithm defined in Section 6 of RFC5280 mandates it, there are actually no perceivable security benefits. While the myriad of string types can accommodate characters from many different languages which might in turn improve human readability, parsing and matching them during the validation of X.509 certificate chain greatly complicates the implementation without concrete security gains. This is especially true when considering the facts that during the programmatic certificate validation, users typically do not read the issuer names.

We argue that some of the rarely used string types can be removed from the standard. For instance, while researching on this topic, we randomly sampled 2 million certificate chains from the Censys [6] data set to gauge the usage number of certain string types. Among the certificate chains we sampled, only 212 chains (0.01%) involve the *teletexString* string type. Recall that *teletexString* is one of the source of confusion in **OpenSSL** (finding #*N* in Table 4). It is understandable that a developer would choose to not support a heavily underutilized string type. Instead of complicating implementations, it might be better to simply remove these string types from the standards.

Instead of $\mathbb{NC}$, a better alternative for chain building would be to simply match the AKI with SKI, the same strategy adopted by **StrongSwan** (see Section 5.3.1). In fact, according to RFC5280, SKI must exist on certificates of conforming CAs, and AKI must exist on certificates issued by conforming CAs. In that case, we recommend fully embracing the mandatory existence of SKI and AKI, and the algorithm defined in Section 6 of RFC5280 can remove $\mathbb{NC}$ as a mandatory requirement, and make matching AKI with SKI mandatory instead. Since the key identifiers in AKI and SKI are basically hashes, they should be easier to parse and match than complex name structures. This can make implementations much simpler. An additional benefit of this recommendation, is that stringprep will be no longer necessary, which helps to reduce the size of the trusted code base, and improve compatibility with resource-constrained platforms. Our current results suggest that no one implements stringprep anyway, and according to library developers, a full-fledged stringprep is indeed a pain point for small footprint implementations.

For $\mathbb{HV}$, our recommendation is that the standards should be updated to explicitly deprecate the use of common name (`CN`) in $\mathbb{HV}$, and to clarify the syntactic (string type and charset) requirements that implementations should enforce. This can help avoid unwarranted leniency from the handling of the complex attribute types and rules associated with `CN`, and allow implementations to tighten their validation decision boundary. We note that at the time of writing, RFC9525 is published, which no longer allows `CN` to be used in $\mathbb{HV}$, and it is supposed to obsolete RFC6125. Our results should motivate support for this new standard. Nevertheless, it appears to us that this simplification and clarification effort can go even further.

In the research and development community, the X.509 PKI is often referred to as the "Web PKI", and X.509 certificates are often called "TLS certificates". This is understandable, as HTTPS/TLS are some of the most widely used applications of X.509. From the perspective of instantiating a PKI for the Web, however, inheriting the historical LDAP and its notion of distinguished names has very limited benefits but greatly complicates the implementations. Distinguished names provide a hierarchical name space useful for LDAP, however, the Web's hierarchical name space today is primarily based on domain names and IP addresses, not LDAP. As such, for applications such as

the Web, there is not much benefit in making subject and issuer names occupy two non-optional certificate fields, and many other attribute/string types in SAN can also be removed. This is especially true as users rarely manually inspect the subject/issuer names, and is simply not possible on many IoT devices that lack displays.

8 Related Works

There is limited prior research on the topic of names and strings in X.509 certificates. The most relevant work in this area is by Sivakorn *et al.* [32], which applies state machine analysis to study the wildcard functionality in TLS libraries. However, wildcard rules are only a subset of the processing and matching requirements associated with names in X.509. The state machine learning attempt does ignore some intricate corner cases concerning string types and charsets. Nonetheless, it demonstrates that discrepancies exist in implementations of $\mathbb{HV}$, especially when it comes to the enforcement of wildcard rules. Additionally, the work by Ma *et al.* [26] reveals that the names of CA certificates are often inaccurate due to lax requirements and other operational complications. This shows that name-related issues can also happen during certificate *issuance*, in addition to the *implementation*-level problems discussed in this paper.

As string processing can be complex but essential to confirming identities, unexpected behaviors could become threats to security. For example, a vulnerability was recently discovered in OpenSSL [14] in its code for parsing CRLs, which was caused by string type confusion. Kumar *et al.* [25] found that CAs sometimes fail to populate subject alternative name extensions with valid DNS names properly, among other issuance problems on the CA side. When internationalized domain names (IDNs) are supported, the potential attack surface might become even wider. Researchers have shown that certain Unicode characters can be misinterpreted, causing victims to visit unintended destinations on the Internet [1].

Much other research has focused on finding bugs in X.509 implementations, using techniques like symbolic execution [7], state machine learning [32], and fuzz testing [4, 11, 27]. Recently there are also some efforts on testing the correctness of the parsing code [9]. However, most of them overlook string-related attributes, and ignore the related processing issues. Other related works examined name checking problems in applications [18, 19], enterprise Wi-Fi [35] and VPNs [34]. There are also some recent attempts in bringing more formal guarantees into X.509 implementations [15, 16], however, the subset of X.509 they target excludes some intricate string types and processing rules.

9 Conclusion

In this study, we employed a portfolio of true-and-tried approaches to test implementations of X.509 name processing, and our investigation achieved better coverage than previous work. We discovered that most implementations do not adhere to the specifications of names in X.509. In particular, some findings can lead to name confusions, where names that should not be matched are considered to match. Our findings provide ample evidence to show that the current standard is unnecessarily complex, so much so that it actually discourages faithful and correct implementations. With the lessons learned from our findings and developers' feedback on our bug reports, we reflect on the design and specification of names in X.509, and recommend possible ways to simplify it.

10 Data Availability

The artifacts, test cases, and additional results of our investigation can be found on GitHub at https://github.com/x509-name-testing/name_testing_artifacts

11 Acknowledgments

We thank the anonymous reviewers for helping us improve our paper. This work was supported in part by an award from the Empire Innovation Program of the New York State, a grant from the Research Grants Council (RGC) of Hong Kong (Project No.: CUHK 24205021), as well as grants from the CUHK IE department (project code: NEW/SYC, GRF/22/SYC, and GRF/23/SYC).

References

[1] Jonathan Birch. 2019. Host/Split: Exploitable Antipatterns in Unicode Normalization. *Black Hat USA 2019* (2019), 1–30. https://www.blackhat.com/us-19/briefings/schedule/#hostsplit-exploitable-antipatterns-in-unicode-normalization-16145

[2] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280. doi:10.17487/RFC5280

[3] Robert T. Braden. 1989. Requirements for Internet Hosts - Application and Support. RFC 1123. doi:10.17487/RFC1123

[4] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy*. IEEE, IEEE, San Jose, CA, USA, 114–129.

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. USENIX Association, 209–224.

[6] Censys. 2024. Censys Dataset. Website. https://censys.com/

[7] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. Symcerts: Practical symbolic execution for exposing noncompliance in X. 509 certificate validation implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE, San Jose, CA, USA, 503–520.

[8] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. 2019. Analyzing semantic correctness with symbolic execution: A case study on pkcs# 1 v1. 5 signature verification. In *Network and Distributed Systems Security (NDSS) Symposium 2019*. The Internet Society, San Diego, CA.

[9] Chu Chen, Pinghong Ren, Zhenhua Duan, Cong Tian, Xu Lu, and Bin Yu. 2023. SBDT: Search-Based Differential Testing of Certificate Parsers in SSL/TLS Implementations. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 967–979.

[10] Chu Chen, Cong Tian, Zhenhua Duan, and Liang Zhao. 2018. RFC-Directed Differential Testing of Certificate Validation in SSL/TLS Implementations. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 859–870. doi:10.1145/3180155.3180226

[11] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 793–804.

[12] Adam M. Costello. 2003. Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA). RFC 3492. doi:10.17487/RFC3492

[13] CVE-2022-3602. 2022. CVE-2022-3602. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-3602.

[14] cve 2023-0286. 2023. cve-2023-0286. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-0286.

[15] Joyanta Debnath, Sze Yiu Chau, and Omar Chowdhury. 2021. On Re-engineering the X. 509 PKI with Executable Specification for Better Implementation Guarantees. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Seoul, Republic of Korea, 1388–1404.

[16] Joyanta Debnath, Christa Jenkins, Yuteng Sun, Sze Yiu Chau, and Omar Chowdhury. 2024. ARMOR: A Formally Verified Implementation of X. 509 Certificate Chain Validation. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 200–200.

[17] eerimoq. 2023. asn1tools. https://github.com/eerimoq/asn1tools.

[18] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 50–61.

[19] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 38–49. doi:10.1145/2382196.2382204

[20] P. Hoffman and M. Blanchet. 2002. Preparation of Internationalized Strings ("stringprep"). RFC 3454 (Proposed Standard). doi:10.17487/RFC3454 Obsoleted by RFC 7564.

[21] UK Statutory Instruments. 2015. The Company, Limited Liability Partnership and Business (Names and Trading Disclosures) Regulations 2015. https://www.legislation.gov.uk/uksi/2015/17/regulation/2.

[22] UK Statutory Instruments. 2015. SCHEDULE 1, The Company, Limited Liability Partnership and Business (Names and Trading Disclosures) Regulations 2015. https://www.legislation.gov.uk/uksi/2015/17/schedule/1.
[23] ITU-T. 2019. https://www.itu.int/rec/T-REC-X.520-201910-I/en.
[24] Ulrich Kühn, Andrei Pyshkin, Erik Tews, and Ralf-Philipp Weinmann. 2008. Variants of Bleichenbacher's Low-Exponent Attack on PKCS#1 RSA Signatures. In *Sicherheit 2008: Sicherheit, Schutz und Zuverlässigkeit. Konferenzband der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 2.-4. April 2008 im Saarbrücker Schloss.*
[25] Deepak Kumar, Zhengping Wang, Matthew Hyder, Joseph Dickinson, Gabrielle Beck, David Adrian, Joshua Mason, Zakir Durumeric, J Alex Halderman, and Michael Bailey. 2018. Tracking certificate misissuance in the wild. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE, San Francisco, CA, USA, 785–798.
[26] Zane Ma, Joshua Mason, Sarvar Patel, Manos Antonakakis, Mariana Raykova, Zakir Durumeric, Phillipp Schoppmann, Michael Bailey, Karn Seth, Sascha Fahl, et al. 2021. What's in a Name? Exploring {CA} Certificate Control. In *30th USENIX Security Symposium (USENIX Security 21)*. 4383–4400.
[27] Lili Quan, Qianyu Guo, Hongxu Chen, Xiaofei Xie, Xiaohong Li, Yang Liu, and Jing Hu. 2020. SADT: syntax-aware differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 524–535.
[28] E. Rescorla. 2000. HTTP Over TLS. RFC 2818 (Informational). doi:10.17487/RFC2818 Updated by RFCs 5785, 7230.
[29] Peter Saint-Andre and Marc Blanchet. 2017. PRECIS Framework: Preparation, Enforcement, and Comparison of Internationalized Strings in Application Protocols. RFC 8264. doi:10.17487/RFC8264
[30] P. Saint-Andre and J. Hodges. 2011. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125 (Proposed Standard). doi:10.17487/RFC6125
[31] Jim Schaad. 2020. *CBOR Object Signing and Encryption (COSE): Header parameters for carrying and referencing X.509 certificates*. Internet-Draft draft-ietf-cose-x509-08. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-cose-x509/08/ Work in Progress.
[32] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. 2017. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE, San Jose, CA, USA, 521–538.
[33] Cong Tian, Chu Chen, Zhenhua Duan, and Liang Zhao. 2019. Differential testing of certificate validation in SSL/TLS implementations: an RFC-guided approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–37.
[34] Ka Lok Wu, Man Hong Hue, Ngai Man Poon, Kin Man Leung, Wai Yin Po, Kin Ting Wong, Sze Ho Hui, and Sze Yiu Chau. 2023. Back to School: On the (In) Security of Academic VPNs. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5737–5754.
[35] Ka Lok Wu, Man Hong Hue, Ka Fun Tang, and Sze Yiu Chau. 2023. The Devil is in the Details: Hidden Problems of Client-Side Enterprise Wi-Fi Configurators. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'23)*. ACM, New York, NY. doi:10.1145/3558482.3590199 (Best Paper Award from ACM WiSec '23).
[36] Moosa Yahyazadeh, Sze Yiu Chau, Li Li, Man Hong Hue, Joyanta Debnath, Sheung Chiu Ip, Chun Ngai Li, Endadul Hoque, and Omar Chowdhury. 2021. Morpheus: Bringing The (PKCS) One To Meet the Oracle. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2474–2496.
[37] K. Zeilenga. 2006. Lightweight Directory Access Protocol (LDAP): Internationalized String Preparation. RFC 4518 (Proposed Standard). doi:10.17487/RFC4518

Received 2024-09-13; accepted 2025-01-14