HVLearn: Automated Black-box Analysis of Hostname Verification in SSL/TLS Implementations

Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana
Department of Computer Science
Columbia University, New York, USA
{suphannee, argyros, kpei, angelos, suman}@cs.columbia.edu

***Abstract*—SSL/TLS is the most commonly deployed family of protocols for securing network communications. The security guarantees of SSL/TLS are critically dependent on the correct validation of the X.509 server certificates presented during the handshake stage of the SSL/TLS protocol. Hostname verification is a critical component of the certificate validation process that verifies the remote server's identity by checking if the hostname of the server matches any of the names present in the X.509 certificate. Hostname verification is a highly complex process due to the presence of numerous features and corner cases such as wildcards, IP addresses, international domain names, and so forth. Therefore, testing hostname verification implementations present a challenging task.**

In this paper, we present HVLearn, a novel black-box testing framework for analyzing SSL/TLS hostname verification implementations, which is based on automata learning algorithms. HVLearn utilizes a number of certificate templates, i.e., certificates with a common name (CN) set to a specific pattern, in order to test different rules from the corresponding specification. For each certificate template, HVLearn uses automata learning algorithms to infer a Deterministic Finite Automaton (DFA) that describes the set of all hostnames that match the CN of a given certificate. Once a model is inferred for a certificate template, HVLearn checks the model for bugs by finding discrepancies with the inferred models from other implementations or by checking against regular-expression-based rules derived from the specification. The key insight behind our approach is that the acceptable hostnames for a given certificate template form a regular language. Therefore, we can leverage automata learning techniques to efficiently infer DFA models that accept the corresponding regular language.

We use HVLearn to analyze the hostname verification implementations in a number of popular SSL/TLS libraries and applications written in a diverse set of languages like C, Python, and Java. We demonstrate that HVLearn can achieve on average 11.21% higher code coverage than existing black/gray-box fuzzing techniques. By comparing the DFA models inferred by HVLearn, we found 8 unique violations of the RFC specifications in the tested hostname verification implementations. Several of these violations are critical and can render the affected implementations vulnerable to active man-in-the-middle attacks.

I. Introduction

The SSL/TLS family of protocols are the most commonly used mechanisms for protecting the security and privacy of network communications from man-in-the-middle attacks. The security guarantees of SSL/TLS protocols are critically dependent on correct validation of X.509 digital certificates presented by the servers during the SSL/TLS handshake phase. The certificate validation, in turn, depends on hostname verification for verifying that the hostname (i.e., fully qualified domain name, IP address, and so forth) of the server matches one of the identifiers in the "SubjectAltName" extension or the "Common Name" (CN) attribute of the presented leaf certificate. Therefore, any mistake in the implementation of hostname verification could completely undermine the security and privacy guarantees of SSL/TLS.

Hostname verification is a complex process due to the presence of numerous special cases (e.g., wildcards, IP addresses, international domain names, etc.). For example, a wildcard character ('*') is only allowed in the left-most part (separated by '.') of a hostname. To get a sense of the complexities involved in the hostname verification process, consider the fact that different parts of its specifications are described in five different RFCs [18], [20], [21], [24], [25]. Given the complexity and security-critical nature of the hostname verification process, it is crucial to perform automated analysis of the implementations for finding any deviation from the specification.

However, despite the critical nature of the hostname verification process, none of the prior research projects dealing with adversarial testing of SSL/TLS certificate validation [36], [38], [45], [50], support detailed automated testing of hostname verification implementations. The prior projects either completely ignore testing of the hostname verification process or simply check whether the hostname verification process is enabled or not. Therefore, they cannot detect any subtle bugs where the hostname verification implementations are enabled but deviate subtly from the specifications. The key problem behind automated adversarial testing of hostname verification implementations is that the inputs (i.e., hostnames and certificate identifiers like common names) are highly structured, sparse strings and therefore makes it very hard for existing black/gray-box fuzz testing techniques to achieve high test coverage or generate inputs triggering the corner cases. Heavily language/platform-dependent white-box testing techniques are also hard to apply for testing hostname verification implementations due to the language/platform diversity of SSL/TLS implementations.

In this paper, we design, implement, and evaluate HVLearn, a black-box differential testing framework based on automata learning, which can automatically infer Deterministic Finite Automata (DFA) models of the hostname verification implementations. The key insight behind HVLearn is that hostname verification, even though very complex, conceptually closely

resemble the regular expression matching process in many ways (e.g., wildcards). This insight on the structure of the certificate identifier format suggests that the acceptable hostnames for a given certificate identifier, as suggested by the specifications, form a regular language. Therefore, we can use black-box automata learning techniques to efficiently infer Deterministic Finite Automata (DFA) models that accept the regular language corresponding to a given hostname verification implementation. Prior results by Angluin et al. have shown that DFAs can be learned efficiently through black-box queries in polynomial time over the number of states [31]. The DFA models inferred by HVLearn can be used to efficiently perform two main tasks that existing testing techniques cannot do well: (i) finding and enumerating unique differences between multiple different implementations; and (ii) extracting a formal, backward-compatible reference specification for the hostname verification process by computing the intersection DFA of the inferred DFA models from different implementations.

We apply HVLearn to analyze a number of popular SSL/TLS libraries such as OpenSSL, GnuTLS, MbedTLS, MatrixSSL, CPython SSL and applications such as Java HttpClient and cURL written in diverse languages like C, Python, and Java. We found 8 distinct specification violations like the incorrect handling of wildcards in internationalized domain names, confusing domain names with IP addresses, incorrect handling of NULL characters, and so forth. Several of these violations allow network attackers to completely break the security guarantees of SSL/TLS protocol by allowing the attackers to read/modify any data transmitted over the SSL/TLS connections set up using the affected implementations. HVLearn also found 121 unique differences, on average, between any two pairs of tested application/library.

The major contributions of this paper are as follows.

- To the best of our knowledge, HVLearn is the first testing tool that can learn DFA models for implementations of hostname verification, a critical part of SSL/TLS implementations. The inferred DFA models can be used for efficient differential testing or extracting a formal reference specification compatible with multiple existing implementations.
- We design and implement several domain-specific optimizations like equivalence query design, alphabet selection, etc. in HVLearn for efficiently learning DFA models from hostname verification implementations.
- We evaluate HVLearn on 6 popular libraries and 2 applications. HVLearn achieved significantly higher (11.21% more on average) code coverage than existing black/gray-box fuzzing techniques and found 8 unique previously unknown RFC violations as shown in Table II, several of which render the affected SSL/TLS implementations completely insecure to man-in-the-middle attacks.

The remainder of this paper is organized as follows: Section II presents the descriptions of the SSL/TLS hostname verification process. We discuss the challenges in testing hostname verification and our testing methodology in Section III. Section IV describes the design and implementation details of HVLearn. We present the evaluation results for using HVLearn to test SSL/TLS implementations in Section V. Section VI presents a detailed case study of several security-critical bugs that HVLearn found. Section VII discusses the related work and Section VIII concludes the paper. For the detailed developer responses on the bugs found by HVLearn, we refer interested readers to Appendix X-B.

II. Overview of hostname verification

As part of the hostname verification process, the SSL/TLS client must check that the host name of the server matches either the "common name" attribute in the certificate or one of the names in the "subjectAltName" extension in the certificate [21]. Note that even though the process is called hostname verification, it also supports verification of IP addresses or email addresses.

In this section, we first provide a brief summary of the hostname format and specifications that describe the format of the common name attribute and subjectAltName extension formats in X.509 certificate. Figure 1 provides a high-level summary of the relevant parts of an X.509 certificate. Next, we describe different parts of the hostname verification process (e.g., domain name restrictions, wildcard characters, and so forth) in detail.

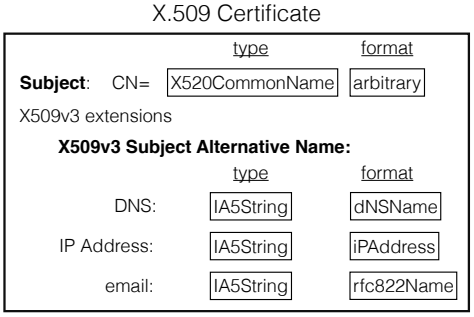


Fig. 1. Fields in an X.509 certificate that are used for hostname verification.

A. Hostname verification inputs

Hostname format. Hostnames are usually either a fully qualified domain name or a single string without any '.' characters. Several SSL/TLS implementations (i.e., OpenSSL) also support IP addresses and email addresses to be passed as the hostname to the corresponding hostname verification implementation.

A domain name consists of multiple "labels", each separated by a '.' character. The domain name labels can only contain letters a-z or A-Z (in a case-insensitive manner), digits 0-9 and the hyphen character '-' [16]. Each label can be up to 63 characters long. The total length of a domain name can be up to 255 characters. Earlier specifications required that the labels must begin with letters [21]. However, subsequent revisions have allowed labels that begin with digits [17].

Common names in X.509 certificates. The Common Name (CN) is an attribute of the "subject distinguished name"

field in an X.509 certificate. The common name in a server certificate is used for validating the hostname of the server as part of the certificate verification process. A common name usually contains a fully qualified domain name, but it can also contain a string with arbitrary ASCII and UTF-8 characters describing a service (e.g., CN='Sample Service'). The only restriction on the common name string is that it should follow the X520CommonName standard (e.g., should not repeat the substring 'CN=') [21]. Note that this is different from the hostname specifications that are very strictly defined and only allow certain characters and digits as described above.

SubjectAltName in X.509 certificates. Subject alternative name (subjectAltName) is an X.509 extension that can be used to store different types of identity information like fully qualified domain names, IP addresses, URI strings, email addresses, and so forth. Each of these types has different restrictions on allowed formats. For example, dNSName(DNS) and uniformResourceIdentifier(URI) must be valid IA5String strings, a subset of ASCII strings [21]. We refer interested readers to Section 4.1.2.6 of RFC 5280 for further reading.

B. Hostname verification rules

Matching order. RFC 6125 recommends SSL/TLS implementations to use subjectAltName extensions, if present in a certificate, over common names as the common name is not strongly tied to an identity and can be an arbitrary string as mentioned earlier [24]. If multiple identifiers are present in a subjectAltName, the SSL/TLS implementations should try to match DNS, SRV, URI, or any other identifier type supported by the implementation and must not match the hostname against the common name of the certificate [24]. The Certificate Authorities (CAs) are also supposed to use the dNSName instead of common name for storing the identity information while issuing certificates [18].

Wildcard in common name/subjectAltName. if a server certificate contains a wildcard character '*', an SSL/TLS implementation should match hostname against them using the rules described in RFC 6125 [24]. We provide a summary of the rules below.

A wildcard character is only allowed in the left-most label. If the presented identifier contains a wildcard character in any label other then the left-most label (e.g., www.*.example.com and www.foo*.example.com), the SSL/TLS implementations should reject the certificate. A wildcard character is allowed to be present anywhere in the left-most label, i.e., a wildcard does not have to be the only character in the left-most label. For example, identifiers like bar*.example.com, *bar.example.com, or f*bar.example.com valid.

While matching hostnames against the identifiers present in a certificate, a wildcard character in an identifier should only apply to one sub-domain and an SSL/TLS implementation should not compare against anything but the left-most label of the hostname (e.g., *.example.com should match foo.example.com but not bar.foo.example.com or example.com).

Several special cases involving the wildcards are allowed in the RFC 6125 only for backward compatibility of existing SSL/TLS implementations as they tend to differ from the specifications in these cases. RFC 6125 clearly notes that these cases often lead to overly complex hostname verification code and might lead to potentially exploitable vulnerabilities. Therefore, new SSL/TLS implementations are discouraged from supporting such cases. We summarize some of them: (i) a wildcard is all or part of a label that identifies a public suffix (e.g., *.com and *.info), (ii) multiple wildcards are present in a label (e.g., f*b*r.example.com), and (iii) wildcards are included as all or part of multiple labels (e.g., *.*.example.com).

International domain name (IDN). IDNs can contain characters from a language-specific alphabet like Arabic or Chinese. An IDN is encoded as a string of unicode characters. A domain name label is categorized as a U-label if it contains at least one non-ASCII character (e.g., UTF-8). RFC 6125 specifies that any U-labels in IDNs must be converted to A-labels domain before performing hostname verification [24]. U-label strings are converted to A-labels, an ASCII-compatible encoding, by adding the prefix 'xn−−' and appending the output of a Punycode transformation applied to the corresponding U-label string as described in RFC 3492 [19]. Both U-labels and A-labels still must satisfy the standard length bound on the domain names (i.e. up to 255 bytes).

IDN in subjectAltName. As indicated in RFC 5280, any IDN in X.509 subjectAltName extension must be defined as type IA5String which is limited only to a subset of ASCII characters [21]. Any U-label in an IDN must be converted to A-label before adding it to the subjectAltName. Email addresses involving IDNs must also be converted to A-labels before.

IDNs in common name. Unlike IDNs in subjectAltName, IDNs in common names are allowed to contain a PrintableString (A-Z, a-z, 0-9, special characters ' = ( ) + , − . / : ?, and space) as well as UTF-8 characters [21].

Wildcard and IDN. There is no specification defining how a wildcard character may be embedded within A-labels or U-labels of an IDN [23]. As a result RFC 6125 [24] recommends that SSL/TLS implementations should not match a presented identifier in a certificate where the wildcard is embedded within an A-label or U-label of an IDN (e.g., xn−−kcry6tjko*.example.com). However, SSL/TLS implementations should match a wildcard character in an IDN as long as the wildcard character occupies the entire left-most label of the IDN (e.g. *.xn−−kcry6tjko.example.com).

IP address. IP addresses can be part of either the common name attribute or the subjectAltName extension (with an 'IP:' prefix) in a certificate. Section 3.1.3.2 of RFC 6125 specifies that an IP address must be converted to network byte order octet string before performing certificate verification [24]. SSL/TLS implementations should compare this octet string with the common name or subjectAltName identifiers. The length of the octet string must be 4 bytes and 18 bytes for IPv4 and IPv6 respectively. The hostname verification should

succeed only if both octet strings are identical. Therefore, wildcard characters are not allowed in IP address identifiers, and the SSL/TLS implementations should not attempt to match wildcards.

Email. Email can be embedded in common name as the emailAddress attribute in legacy SSL/TLS implementations. The attribute is not case sensitive. However, new implementations must add email addresses in rfc822Name format to subject alternative name extension instead of the common name attribute [21].

Internationalized email. As similar to IDNs in subjectAltName extensions, an internationalized email must be converted into the ASCII representation before verification. RFC 5321 also specifies that network administrators must not define mailboxes (local-part@domain/address-literal) with non-ASCII characters and ASCII control characters. Email addresses are considered to match if the local-part and host-part are exact matches using a case-sensitive and case-insensitive ASCII comparison respectively (e.g., MYEMAIL@example.com does not match myemail@example.com but matches MYEMAIL@EXAMPLE.COM) [21]. Note that this specification contradicts that of the email addresses embedded in the common name that is supposed to be completely case-insensitive.

Email with IP address in the host part. RFCs 5280 and 6125 do not specify any special treatment for IP address in the host part of email and only allow email in rfc822Name format. The rfc822Name format supports both IPv4 and IPv6 addresses in the host part. Therefore, an email with an IP address in the host part is allowed to be present in a certificate [22].

Wildcard in email. There is no specification that wildcard should be interpreted and attempted to match when they are part of an email address in a certificate.

Other identifiers in subjectAltName. There are other identifiers that can be used to perform identity checks e.g., UniformResourceIdentifier(URI), SRVName, and otherName. However, most popular SSL/TLS libraries do not support checking these identifiers and leave it up to the applications.

III. Methodology

In this section, we describe the challenges behind automated testing of hostname verification implementations. Albeit small in size, the diversity of these implementations and the subtleties in the hostname verification process make these implementations difficult to test. We then proceed to describe an overview of our methodology for testing hostname verification implementations using automata learning algorithms. We also provide a brief summary of the basic setting under which automata learning algorithms operate.

A. Challenges in hostname verification analysis

We believe that any methodology for automatically analyzing hostname verification functionality should address the following challenges:

1. Ill-defined informal specifications. As discussed in Section II, although the relevant RFCs provide some examples/rules defining the hostname verification process, many corner cases are left unspecified. Therefore, it is necessary for any hostname verification implementation analysis to take into account the behaviors of other popular implementations to discover discrepancies that could lead to security/compatibility flaws.

2. Complexity of name checking functionality. Hostname verification is significantly more complex than a simple string comparison due to the presence of numerous corner cases and special characters. Therefore, any automated analysis must be able to explore these corner cases. We observe that the format of the certificate identifier as well as the matching rules closely resemble a regular expression matching problem. In fact, we find that the set of accepted hostnames for each given certificate identifier form a regular language.

3. Diversity of implementations. The importance and popularity of the SSL/TLS protocol resulted in a large number of different SSL/TLS implementations. Therefore, hostname verification logic is often implemented in a number of different programming languages such as C/C++, Java, Python, and so forth. Furthermore, some of these implementations might be only accessible remotely without any access to their source code. Therefore, we argue that a black-box analysis algorithm is the most suitable technique for testing a large variety of different hostname verification implementations.

B. HVLearn's approach to hostname verification analysis

Motivated by the challenges described above, we now present our methodology for analyzing hostname verification routines in SSL/TLS libraries and applications.

The main idea behind our HVLearn system is the following: For different rules in the RFCs as well as for ambiguous rules which are not well defined in the RFC, we generate "template certificates" with common names which are specifically designed in order to check a specific rule. Afterward, we use automata learning algorithms in order to extract a DFA which describes the set of all hostname strings which are matching the common name in our template certificate. For example, the inferred DFA from an implementation for the identifier template "aaa.*.aaa.com" can be used to test conformance with the rule in RFC 6125 prohibiting wildcard characters from appearing in any other label than the leftmost label of the common name.

Once a DFA model is generated by the learning algorithm, we check the model for violations of any RFC rules or for other suspicious behavior. HVLearn offers two methods to check an inferred DFA model:

Regular-expression-based rules. The first option allows the user to provide a regular expression that specifies a set of invalid strings. HVLearn can ensure that the inferred DFAs do not accept any of those strings. For example, RFC 1035 states that only characters in the set [A-Za-z0-9] and the characters '-' and '.' should be used in hostname identifiers. Users therefore can construct a simple regular expression that can be used by HVLearn to check whether any of the tested implementations accept a hostname with a character outside the given set.

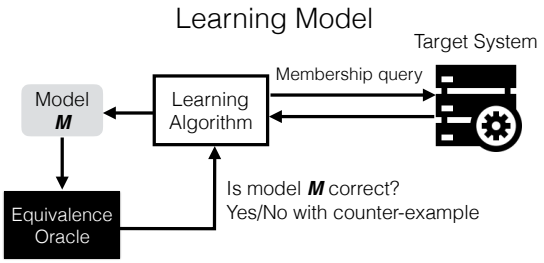


Fig. 2. *Exact learning from queries:* the active learning model under which our automata learning algorithms operate.

Differential testing. The second option offered by HVLearn is to perform a differential testing between the inferred model and models inferred from other implementations for the same certificate template. Given two inferred DFA models, HVLearn generates a set of *unique differences* between the two models using an algorithm which we discuss in Section IV-E. This option is especially useful for finding bugs in corner cases which are not well defined in the RFCs.

We summarize the advantages of our approach below:

- Adopting a black-box learning approach ensures that our analysis method is language independent and we can easily test a variety of different implementations. Our only requirement is the ability to query the target library/application with a certificate and a hostname of our choice and find whether the hostname is matching the given identifier in the certificate.
- As pointed out in the previous section, hostname verification is similar to regular expression matching. Given that regular expressions can be represented as DFAs, adopting an automata-based learning algorithm for representing the inferred models for each certificate template is a natural and effective choice.
- Finally, an additional advantage of having DFA models is that we can efficiently compare two inferred models and enumerate all differences between them. This property is very important for differential testing as it helps us in analyzing the ambiguous rules in the specifications.

Limitations. A natural trade-off of choosing to implement our system as a black-box analysis method is that we cannot guarantee completeness or soundness of our models. However, each difference inferred by HVLearn can be easily verified by querying the corresponding implementations. Moreover, since our system will find all differences among implementations, it will not report a bug that is common among all implementations unless a rule is explicitly specified for it, as described above. Finally, we point out that not all discrepancies among systems are necessarily security vulnerabilities; they may represent equally acceptable design choices for ambiguous parts of the RFCs.

C. Automata Learning Algorithms

We will now describe the automata learning algorithms that allow us to realize our automata-based analysis framework.

Learning model. We utilize learning algorithms that work in an active learning model which is called *exact learning from queries*. Traditional supervised learning algorithms, such as those used to train deep neural networks, work on a given set of labeled examples. In contrast, active learning algorithms in our model work by adaptively selecting inputs that they use to query a target system and obtain the correct label.

Figure 2 presents an overview of our learning model. A learning algorithm attempts to learn a model of a target system by querying the target system with inputs of its choice. Eventually, by querying the target system multiple times, the learning algorithm infers a model of the target system. This model is then checked for correctness through an *equivalence oracle*, an oracle that checks whether the inferred model correctly summarizes the behavior of the target system. If the model is correct, i.e., it agrees with the target system on all inputs, then the learning algorithm will output the generated model and terminate. On the other hand, if the model is incorrect, the equivalence oracle will produce a *counterexample*, i.e., an input under which the target system and the model produce different outputs. The learning algorithm then uses the counterexample to refine the inferred model. This process iterates until the learning algorithm produces a correct model.

To summarize, a learning algorithm in the exact learning model is able to interact with the target system using two types of queries:

- **Membership queries:** The input to this type of query is a string s and the output is `Accept` or `Reject` depending on whether the string s is accepted by the target system or not.
- **Equivalence queries:** The input to an equivalence query is a model M and the output of the query is either $True$, if the model M is equivalent to the target system on all inputs, or a counterexample input under which the model and target system produce different outputs.

Automata learning in practice. The first algorithm for inferring DFA models in the exact learning from queries model was developed by Angluin [31] and was followed by a large number of optimizations and variations in the following years. In our system, we use the Kearns-Vazirani (KV) algorithm [54]. The KV algorithm utilizes a data structure called the discrimination tree and it is in practice more efficient in terms of the amount of queries it requires to infer a DFA model.

The most significant challenge that one should address in order to use the KV algorithm and other automata learning algorithms in practice, is how to implement an efficient and accurate equivalence oracle in order to simulate the equivalence queries performed by the learning algorithm. Since we only have black-box access to the target system, any method for implementing equivalence queries is necessarily incomplete.

In HVLearn, we use the Wp-method [49], for implementing equivalence queries. The Wp-method checks the equivalence between an inferred DFA and a target system using only black-box queries to the target system. Essentially, the Wp-method approximates an equivalence oracle by using multiple

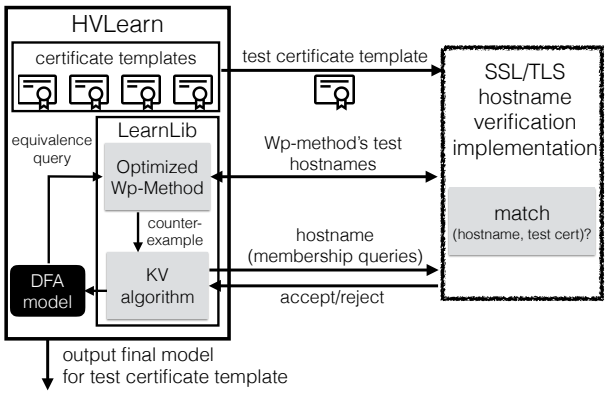


Fig. 3. Overview of learning a hostname verification implementation using HVLearn.

membership queries. The algorithm is given as input the DFA to be checked and an upper bound on the number of states in the target system when modeled as a DFA, a parameter which we call *depth*. Then, the algorithm creates a set of test inputs S, which are then submitted to the target system. If the target system agrees with the DFA model on all inputs in the test set S, then the DFA and the target system are proved equivalent under the assumption that the upper bound on the number of states of the target system is correct.

In theory, one can set the depth parameter of the Wp-method to a very large value in order to design an equivalence oracle which is, in practice, complete. However, the size of the set of test inputs produced by the Wp-method is on the order of $O(n^2|\Sigma|^{m-n+1})$ where Σ is the input alphabet for the DFA, m is the upper bound on the number of states of the target system and n is the number of states in the input DFA. Therefore, using the Wp-method with a large depth (i.e., upper bound on the number of states of the target system) is impractical. Note that, the bound on the number of test inputs produced by the Wp-method is not a worst case bound; on the contrary, the number of test inputs produced is usually of that order.

Consequently, it is essential for the efficiency of our system to maintain a small alphabet for our DFAs and also set a small upper bound (depth) on the number of states of the target system while using the Wp-method. We address both of these issues in the next section.

IV. Architecture of HVLearn

In this section, we describe the design and implementation of our system, HVLearn, based on automata learning techniques. Specifically, we describe the technical challenges that arise when we attempt to use automata learning algorithms in practice. We also summarize the optimizations that HVLearn implements to address these challenges and efficiently learn DFA models of hostname verification implementations.

A. System overview

Figure 3 presents an overview of how HVLearn is used to analyze the hostname verification functionality of an SSL/TLS library. To use HVLearn, the user provides HVLearn access to the hostname verification function that takes an X.509 certificate and a hostname as input and returns `accept/reject` depending on whether the provided hostname is matching the identifier in the certificate. We describe how we implement this interface in Section IV-C. Our system includes a number of certificate templates, which are certificates designed to test the SSL/TLS implementation on a number of different rules as described in Section IV-B. For each such template, HVLearn will learn a DFA model describing the set of hostnames accepted by a given implementation for the given certificate template. To produce a DFA model, HVLearn utilizes the LearnLib [59] library which contains implementations of both the KV algorithm and the Wp-method. To avoid setting the maximum depth of the Wp-method to impractically high values, we optimize the equivalence oracle as described in Section IV-D.

Once a model is generated, our system proceeds to analyze the model as described in Section IV-E. The results of our analysis, both the inferred models and the differences between models are then saved for reuse. Optionally, HVLearn can also utilize the inferred models for a certificate template to extract a formal specification for the corresponding certificate template as described in Section V-F.

B. Generating certificate templates

To cover all different rules and ambiguous practices in hostname verification, we created a set of 23 certificates with different identifier templates, where each certificate is designed to test a specific rule from the specification. These certificates are selected to cover all the rules we described in Section II. For example, a certificate with common name "xn−−a*.aaa" will test if the implementation allows wildcards as part of an A-label in an IDN, something which is explicitly forbidden by RFC 6125. Our template certificates are self-signed X.509 v3 certificates generated using the GnuTLS library. We choose to use GnuTLS for certificate generation because it allows identifiers with embedded NULL characters in both subject common name and SAN. The template identifier to be tested is placed in either Subject CN and/or SAN (as dNSName, iPAddress, or email).

C. Performing membership queries

In order to utilize the learning algorithms in LearnLib (including the Wp-method), we implement a *membership query* function that performs all queries to the target system. This function accepts input as a string and returns a binary value. In our system, we use the hostname verification function from the target SSL/TLS implementation. We note here that, since LearnLib is written in Java while many of our tested SSL/TLS implementations are written in C/C++/Python, we utilized the Java Native Interface (JNI) [10] to efficiently perform membership queries to the target in such cases.

D. Automata learning parameters and optimizations

In this section, we describe the architectural decisions and optimizations that we implemented to efficiently scale the KV

algorithm for testing complex real-world SSL/TLS hostname verification implementations.

Alphabet size. The first important decision we have to make to utilize the KV algorithm is to select an alphabet that will be used by the algorithm. The alphabet refers to the set of symbols that the learning algorithm will test.

A straightforward approach is to use a very general set of characters such as the set of ASCII characters. However, this will impose an unnecessary overhead in our system's performance since the performance of both the KV algorithm and the Wp-method rely heavily on the underlying alphabet size. Our main insight is that we can reduce the alphabet to a small set of representative characters that will thoroughly test all different aspects of hostname verification. In particular we select the set $\Sigma = \{$ a, 1, dot, \s, @, A, =, *, x, n, -, \u4F60, NULL$\}$ as an input alphabet in our experiments. In the presented alphabet, 'dot' denotes the '.' character, \s denotes the space character (ASCII value 32), NULL denotes the zero byte character, and \u4F60 denotes the unicode character with hexadecimal value 4F60.

Note that this set of symbols is adequate for analyzing hostname verification implementations since it includes characters from all different categories such as lowercase, uppercase, digits, unicode, etc., as well as special characters like the NULL character. The lowercase characters 'x', 'n' in conjunction with the '-' character are necessary in order to encode IDN hostnames. Finally, the inclusion of some non-alphanumeric characters such as the '=' character allows us to detect violations where an implementation accepts invalid hostnames.

Note that, even though the hostnames generated using this alphabet set will often not resolve to a real IP address when processed as DNS names, it does not affect the accuracy of our analysis in any way. This is a side-effect the fact that the hostname verification routines are not responsible for resolving the provided DNS name to an IP address. It simply checks whether the given hostname matches the identifier in the provided certificate.

Caching membership queries. To avoid the communication cost of repeated querying of the SSL/TLS implementations with same inputs, we utilize LearnLib's `DFALearningCache` class to cache the results of the membership queries. The cache is checked on each new query, and a cached result is used whenever found. This optimization is particularly useful for cutting down the overhead of the repeated queries generated by the Wp-method across multiple equivalence queries.

Optimizing equivalence queries. In practice, the first model generated by the learning algorithm is usually just single state DFA which rejects all hostnames. The reason is that the learning algorithm is not able to generate any accepting hostname and thus cannot distinguish between the initial state and any other state in the target system. Sometimes, to force the KV algorithm to produce an accepting hostname using the Wp-method, a very large depth is required. This may cause efficiency issues in the system. However, if we supply the model with an accepting hostname, then trivial models will be improved quickly without having to utilize excessive depth parameters in the Wp-method.

Recall here that the exponential term in the Wp-method is dependent on the difference between the number of states in the model and the provided depth. Therefore, once we discover an accepting state in the target system, the Wp-method with a much smaller depth will still be able to explore many different aspects of the hostname verification implementation.

In order to generate an accepting hostname, we perform the following test during an equivalence query and before calling the Wp-method. First, we search for any wildcard characters (*) in the provided common name and replace them with random characters from our alphabet to obtain a concrete hostname. Next, we check that the generated model and the target hostname verification implementation agree on a set of hostnames generated using this method. If not, we return the hostname for which they differ as a counterexample. The main advantage of this heuristic is that it allows us to quickly produce accepting hostnames that uncover new states in the target system without invoking the Wp-method with very large depth values. Once these states are uncovered, and the quality of the inferred models improve, the Wp-method, with a small depth parameter, is utilized to discover additional states in the target system.

E. Analysis and comparison of inferred DFA models

After HVLearn outputs a model, the next task for our system is to analyze the produced model for RFC violations or, confusing/ambiguous rules in the RFC, to compare different inferred models and analyze any discrepancies found between different implementations.

Analyzing a single DFA model. In the case of a single model, we would like to determine whether the model is accepting invalid hostnames prohibited by the RFC specification. If the specification is unclear, our analysis can still be used in order to manually inspect the behavior of the implementation on the specific certificate template besides the differential analysis described below.

Our system offers two options for performing analysis of a single model. First, our system generates inputs that will exercise all simple paths (i.e., paths without loops) that lead to accepting states, in the inferred model. Intuitively, these inputs are a small set of inputs that describe all different flavors of hostnames that will be accepted for the given certificate template. By inspecting these certificates, we can determine if the implementation is accepting invalid hostnames. Second, HVLearn allows the user to specify a regular expression rule to be checked against the inferred model. In this case, the user specifies a regular expression and HVLearn verifies that the regular expression and the inferred model does not share any common strings. This option allows to easily check certain RFC violations by utilizing simple regular expression rules. For example, consider the rule specifying that no non-alphanumeric characters should be part of a matching hostname. By specifying the regular expression rule "(.)*=(.)*"

we can check whether there exists any matching hostname that contains the '=' character in the inferred model.

Comparing unique differences between DFA models. For analyzing certain corner cases which are not specified in the RFC, testing a single model may not be enough. Instead, we compare the inferred models for different SSL/TLS implementations and find inputs under which the implementations behave differently. To perform this analysis, we utilize the difference enumeration algorithm from [33]. In a nutshell, this algorithm computes the product DFA between two, or more, given models and then finds all simple paths to states in which the DFAs are producing different output.

F. Specification Extraction

As we discussed already, the RFC specifications leave certain aspects of hostname verification up to the implementations by not specifying the correct behavior in all cases. In these cases imposing specific restrictions in the implementations is challenging since we have to be careful to avoid breaking compatibility with existing implementations and valid certificates. In this section, we describe how the inferred DFA models for the different certificate templates can be used to infer a formal specification, which is compatible with existing implementations, for the cases where RFC specifications are vague.

Our main insight is the following: *For each certificate template, we can use the DFA accepting the set of hostnames accepted by all SSL/TLS implementations as a formal specification of the corresponding rule template.* The intuition behind this choice is that this specification is avoiding small idiosyncrasies of each library and it is thus very compact. On the other hand, if a vulnerability exists in this specification then this vulnerability must also exist in all tested implementations. Since each implementation is audited independently, our choice gives us confidence that our specification is secure from simple vulnerabilities while maintaining backward compatibility with the tested implementations.

Computing the specification. In order to compute the corresponding specification for each certificate template, we proceed as follows: First, we obtain DFA models for all hostname verification implementations under test using HVLearn. Next, we compute the product DFA for all the inferred models. The product DFA accepts the intersection of the regular languages of each DFA. We compute the product DFA using standard automata algorithms [60]. The inferred formal specification for our set of implementations is represented by the product DFA of each DFA model. This product DFA can be then converted back to a regular expression to improve readability.

Finally, we would like to point out that computing the intersection of k DFAs have a worst case time complexity of $O(n^k)$ where n is the number of states in each DFA [55]. However, in our case, the inferred DFAs are mostly similar and thus, the product construction is very efficient because intersecting two DFAs is not adding a significant number of states in the resulting product DFA. We provide more evidence supporting this hypothesis in Section V.

V. Evaluation

The main goals of our evaluation of HVLearn to answer the following questions: (i) how effective HVLearn is in finding RFC violations in real-world hostname verification implementations? (ii) How much do our optimizations help in improving the performance of HVLearn? (iii) how does HVLearn perform compare to existing black-box or coverage-guided gray-box techniques (iv) can HVLearn infer backward-compatible specifications from the inferred DFAs of real-world hostname verification implementations.

A. Hostname verification test subjects

We use HVLearn to test hostname verification implementations in six popular open-source SSL/TLS implementations, namely OpenSSL, GnuTLS, MbedTLS (PolarSSL), MatrixSSL, JSSE, and CPython SSL, as well as in two popular SSL/TLS applications: cURL and HttpClient. Note that as several libraries like OpenSSL versions prior to 1.0.1 do not provide support for hostname verification and leave it up to the application developer to implement it. Therefore, applications like cURL/HttpClient that support different libraries are often forced to write their own implementations of hostname verification.

Among the libraries that support hostname verification, some like OpenSSL provide separate API functions for matching each type of identifier (i.e., domain name, IP addresses, email, etc.) and leave it up to application to select the appropriate one depending on the setting. In contrast, others like MatrixSSL combine all supported types of identifiers in one function and figure out the appropriate by inspecting the input string. Table I shows the hostname verification function/class names for all implementations that we tested and the types of identifier(s) that each of them supports. The last column shows physical source lines of code (SLOC) for each host matching function/class as reported by the SLOCCount [14] tool. Note that the shown SLOC only count the parts of the code that perform hostname matching.

B. Finding RFC violations with HVLearn

We use HVLearn to produce DFA models for each distinct certificate template corresponding to different patterns from the RFCs. Afterward, we detect potentially buggy behavior by both performing differential testing of output DFAs as well as checking individual DFAs for violations of regular-expression-based rules that we created manually as described in Section IV-E.

Table II presents the results of our experiments. We evaluated a diverse set of rules from four different RFCs [16], [17], [21], [24]. We found that every rule that we tested is violated by at least one implementation, while on average each implementation is violating three RFC rules. Several of these violations have severe security implications (e.g., mishandling wildcard characters in international domain names, confusing IP addresses as domain names etc.). We describe these cases along with their security implications in detail in Section VI.

TABLE I
HOSTNAME VERIFICATION FUNCTIONS (ALONG WITH THE TYPES OF SUPPORTED IDENTIFIERS) IN SSL/TLS LIBRARIES AND APPLICATIONS

SSL/TLS Libs/Apps	Version	Supported Identifier(s)	Hostname Matching Function/Class Name	Approx. SLOC
OpenSSL	$\leqslant$ 1.0.1	–	–	–
OpenSSL	$\geqslant$ 1.0.2	CN/DNS	X509_check_host	314
		IP	X509_check_ip	308
		IP	X509_check_ip_asc	417
		EMAIL	X509_check_email	314
GnuTLS	3.5.3	CN/DNS/IP	gnutls_x509_crt_check_hostname, gnutls_x509_crt_check_hostname2	195
		EMAIL	gnutls_x509_crt_check_email	149
MbedTLS	2.3.0	CN/DNS	mbedtls_x509_crt_verify, mbedtls_x509_crt_verify_with_profile	193
MatrixSSL	3.8.4	CN/DNS/IP/ EMAIL	matrixValidateCerts	130
JSSE	1.8	CN/DNS/IP	HostnameChecker	202
CPython SSL	3.5.2	CN/DNS/IP	match_hostname	59
HttpClient	4.5.2	CN/DNS/IP	DefaultHostnameVerifier	257
cURL	7.50.3	CN/DNS/IP	verifyhost, Curl_verifyhost	300

Note that the library with the most violations is JSSE (four violations), while HttpClient is the application with the most violations (five violations). OpenSSL, MbedTLS, and CPython SSL only have two violations each, having common the violation of matching invalid hostnames. The interested reader can find an extended description of our results in the Appendix (Table VIII).

C. Comparing unique differences between DFA models

In order to evaluate the discrepancies between all different hostname verification implementations, we computed the number of differences for each pair of hostname verification implementations in our test set. Recall that for two given DFA models we define the number of differences as the number of simple paths in the product DFA which lead to a different output being produced by the two models [33].

Table III presents the results of our experiment. For example, OpenSSL and GnuTLS have 95 discrepancies in total. This is obtained by summing up the number of unique paths that are different between the inferred DFAs for each common name in Table VIII. Note that all pairs of implementations contain a large number of unique cases under which they produce a different output. As seen in Table III, each pair of tested implementation has 127 unique differences on average between them. We note that some differences only imply ambiguous RFC rules while some reveal the potential invalid hostnames or RFC violation bugs. The interested reader can find a more detailed list of the unique strings that each implementation is accepting in Table VIII in the Appendix. In any case, we find the fact that all implementations of such a security critical component of the SSL/TLS protocol present such a larger number of discrepancies to be an alarming issue since it signifies either a poor implementation of the specification or vagueness in the specification itself. Our analysis suggests that both cases are present in practice.

D. Comparing code coverage of HVLearn and black/gray-box fuzzing

In order to compare HVLearn's effectiveness in finding bugs with that of black/gray-box fuzzing, we investigate the following research question:

RQ.1: How HVLearn's code coverage differ from black/gray-box fuzzing techniques?

We compare the code coverage of the tested hostname verification implementations achieved by HVLearn and two other techniques, black-box fuzzing, and coverage-guided gray-box fuzzing. We describe our testing setup briefly below.

HVLearn: HVLearn leverages automata learning that invokes the hostname verification matching routine with a predefined certificate template and alphabet set. HVLearn adaptively refines a DFA corresponding to the test hostname verification implementation by querying the implementation with new hostname strings. We measure the code coverage achieved during the learning process until it finishes. We also monitor the total number of queries NQ, which comes from both the membership and the equivalence queries.

Black-box fuzzing: With the same alphabet and certificate template used by HVLearn, we randomly generate NQ strings and query the target SSL/TLS hostname verification function with the same certificate template. Note that the black-box fuzzer generates independent random strings without any sort of guidance.

Coverage-guided gray-box fuzzing: Unlike black-box fuzzing, coverage-guided gray-box fuzzing tries to generate more interesting inputs by using evolutionary techniques to the input generation process. In each generation, a new batch of inputs are generated from the previous generation through mutation/cross-over and only the inputs that increase code coverage are kept for further changes. Coverage-guided gray-box fuzzing is a popular technique for finding bugs in large real-world programs [6], [11].

To make it a fair comparison with HVLearn, we implemented our own coverage-guided gray-box fuzzer as existing tools like AFL do not provide an easy way of restricting the mutation outputs within a given alphabet. With the same alphabet set, we initialize the fuzzer with a set of strings of varying lengths as the seeds maintained in a queue Q. The seeds are then used by the fuzzer to query the target hostname verification implementation. After finishing querying, using the seeds, the fuzzer gets the string $S = dequeue(Q)$. It randomly mutates one character within S and obtains S'. Then it uses the mutated S' to query the target. If the mutated string S' increased code coverage, we store it in the queue for further mutation, i.e., $enqueue(S', Q)$. Otherwise, we throw it away. The fuzzer is thus guided to always mutate on the strings that have better code coverage. The fuzzer iteratively performs this enqueue/dequeue operations for NQ rounds, and we obtain the final code coverage COV_{randmu} of each

TABLE II
A SUMMARY OF RFC VIOLATIONS AND DISCREPANT BEHAVIORS FOUND BY HVLEARN IN THE TESTED SSL/TLS LIBRARIES AND APPLICATIONS

RFC Violations	RFC	OpenSSL	GnuTLS	MbedTLS	MatrixSSL	JSSE	CPython SSL	cURL	HttpClient	HttpClient*
Invalid hostname character										
Only alphanumeric and '-' matches in hostname	1035	✗	✗	✗	✓	✗	✗	✗	✗	✗
Case-insensitive hostname										
Match CN in case-insensitive manner	5280, 6125	✓	✓	✓	✓	✓	✓	✓	✗	✗
Wildcard										
Not attempt to match wildcard not in left-most label (CN/DNS: aaa.*.aaa)	6125	✓	✓	✓	✓	✗	✓	✓	✗	✓
IDN and wildcard										
Not attempt to match wildcard fragment in IDN (xn−−a*.aaa)	6125	✓	✓	✓	✓	✗	✓	✓	✗	✓
Common name and subjectAltName										
No CN checked when DNS presents	6125	✓	✓	✓	✗	✓	✓	✓	✓	✓
No CN checked when any SAN ID presents	6125	–	✗	–	✗	✗	✓	✓	✗	✗
Email-based certificate										
Case-sensitive on local-part of email attribute in SAN	5280	✓	✓	–	✗	–	–	–	–	–
IP address-based certificate										
Not attempt to match IP address with DNS (DNS: 1.1.1.1)	1123	–	✗	✗	✗	✓	✓	✓	✓	✓
Discrepancies										
Wildcard										
Attempt to match wildcard with empty label (hostname: .aaa.aaa with CN/DNS: *.aaa.aaa)	–	✓	✓	✗	✗	✗	✗	✗	✓	✓
Attempt to match wildcard in public suffix (CN/DNS: *.co.uk)	6125	✓	✗	✓	✓	✓	✓	✓	✓	✗
Embedded NULL character										
Allowed NULL character in CN	–	✓	✓	✓	✗	✓	✓	✓	✓	✓
Allowed NULL character in SAN	–	✓	✓	✗	✗	✓	✓	✓	✓	✓
Match NULL character hostname: b.b\0.a.a, CN/DNS: b.b\0.a.a	–	✗	✗	✗	✗	✓	✓	✗	✓	✓
Other invalid hostname										
Partially match suffix (hostname: .a with CN/DNS: a.a, a.a.a)	1035	✓	✗	✗	✗	✗	✗	✗	✗	✗
Match trailing dot (hostname: aaa.aaa with CN/DNS: aaa.aaa)	–	✗	✗	✗	✗	✗	✗	✓	✗	✗

HttpClient*: HttpClient with `PublicSuffixMatcher`
For RFC Violation: ✓= OK, ✗= RFC violate, – = libs/apps do not support • For Discrepancies: ✓= Accept, ✗= Reject

TABLE III
NUMBER OF UNIQUE DIFFERENCES BETWEEN AUTOMATA INFERRED FROM DIFFERENT SSL/TLS IMPLEMENTATIONS

	OpenSSL	GnuTLS	MbedTLS	MatrixSSL	JSSE	CPython	HttpClient	Curl
OpenSSL	–	95	98	99	282	92	482	187
GnuTLS	–	–	6	38	127	34	214	56
MbedTLS	–	–	–	44	97	28	220	50
MatrixSSL	–	–	–	–	37	25	58	94
JSSE	–	–	–	–	–	69	177	110
CPython	–	–	–	–	–	–	108	54
HttpClient	–	–	–	–	–	–	–	414
Curl	–	–	–	–	–	–	–	–

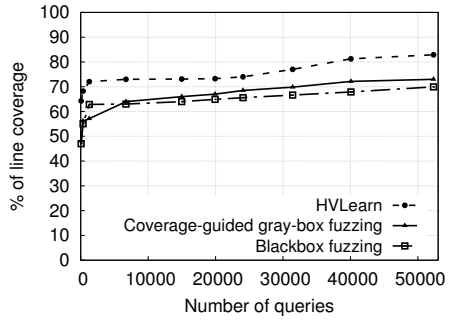


Fig. 4. Comparison of code coverage achieved by HVLearn, gray-box fuzzing, and black-box fuzzing for OpenSSL hostname verification.

functions SSL/TLS implementations. Note that we keep the test certificate template fixed during the entire test.

We use the percentage of lines executed, which are extracted by Gcov [51], as the indicator for the code coverage. Considering that hostname verification is a small part of an SSL/TLS implementation, we do not compute the percentage of lines covered with respect to the total number of lines. Instead, we calculate the percentage of line coverage within each function and only take into account the functions that are related to hostname verification.

> ***Result 1:*** *HVLearn achieves 11.21% increase in code coverage on average when comparing to the black/gray-box fuzzing techniques.*

Therefore, let LE(f) be the number of lines executed of function f in the SI and L(f) be the total number of lines of f, the code coverage can be defined in the following equa-

tion: $coverage = \frac{\sum_{i=1}^{m} LE(f_i)}{\sum_{i=1}^{m} L(f_i)}$, where $f_1, f_2, \cdots, f_m$ are the functions that are relevant to hostname verification. Figure 4 illustrates the code coverage comparison, which shows that HVLearn achieves significantly better code coverage compared to the black/gray-box fuzzing techniques.

E. Automata learning performance

HVLearn is largely based on the KV algorithm and the Wp-method in order to perform its analysis. It is therefore crucial to thoroughly evaluate the different parameters of these algorithms and their impact on the performance of HVLearn. We will now evaluate the effect of each different parameter of the learning algorithms in the overall performance of HVLearn.

RQ.2: How does the alphabet size affect HVLearn's performance in practice?

As discussed in Section III-C, the alphabet size impacts the performance of our system. In theory, the performance of both the KV algorithm and the Wp-method, depends on the size of the input alphabet. We perform two experiments for evaluating the extent to which the alphabet size affects the performance of our learning algorithm component in practice. In the first experiment, we evaluate the effect of increasing the size of the alphabet in real world DNS names. For this experiment, we used our system in the default configuration with all optimizations (e.g., query cache and EQ optimizations) enabled and we set the Wp-method depth to 1. We used the CPython's SSL implementation as the hostname verification function for these experiments.

Figure 5 shows the results of our experiment. Notice that, starting from an alphabet size of 9, each additional character we include in the alphabet will cause the learning algorithm to perform at least 10% more queries in order to produce a model, for both DNS names, while this percentage is only increasing when in larger alphabet sizes.

We also measure the effect of increasing the alphabet size on the overall running time of our system. To perform this experiment we used the same setup as our previous experiment and evaluated the performance of HVLearn with a certificate containing the common name "*.aaa.aaa". Table IV shows the results of this experiment. We notice that the increase in the membership queries directly translates in an increased running time. Specifically, by adding 5 additional characters in the alphabet (from 2 to 5), we notice that the running time increases 7 times. Similar results can be observed when we add more characters in the alphabet set.

> ***Result 2:*** *Adding just one symbol in the alphabet set incurs at least 10% increase in the number of queries. Thus, the succinct alphabet set utilized by HVLearn is crucial for the system's performance.*

RQ.3: Does membership cache improve the performance of HVLearn?

Table IV presents the number of queries required to infer a model for the certificate template with common name

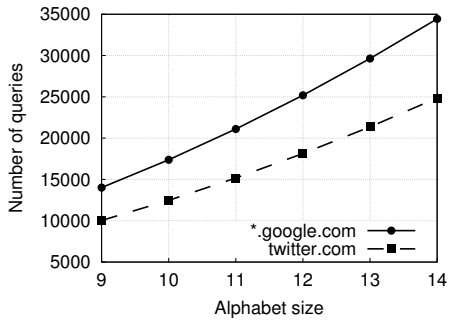


Fig. 5. Number of queries required to learn an automaton with different alphabet sizes (with Wp-method depth=1 and equivalence query optimization).

TABLE IV
HVLEARN PERFORMANCE FOR COMMON NAME `*.AAA.AAA` WITH WP-METHOD DEPTH=1 (CPYTHON SSL IMPLEMENTATION)

Alphabet Size	W/o Cache #Queries	With Cache #Queries				Average Time (sec)
	Total	Total	Membership	Equivalence		
				Counterexample	Membership	
2	883	226	136	2	90	3.10
5	3,049	1,582	436	2	1,146	21.61
7	5,163	3,156	636	2	2,520	42.24
10	9,339	6,522	936	2	5,586	86.92
15	18,979	14,812	1,436	2	13,376	196.35

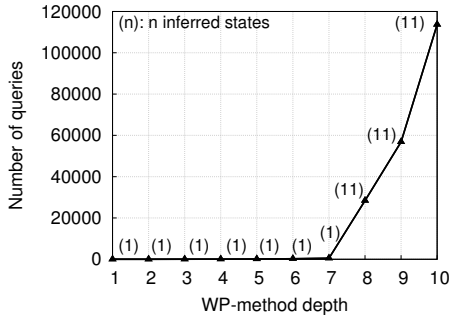


Fig. 6. The number of queries needed to learn the DFA model of CPython certificate verification for different Wp-method depth values (without equivalence query optimization).

"*.aaa.aaa" with and without utilizing a membership query cache over different alphabet sizes. We notice that the cache is consistently helping to reduce the number of membership queries required to infer a model. Overall, the cache is reducing the number of queries by 42%, thus significantly improving the efficiency of our system. Therefore, for the rest of the experiments in this section, we utilize our system with the membership query cache enabled.

> ***Result 3:*** *Membership cache is offering, on average, a 42% decrease on the number of membership queries made by the learning algorithm.*

RQ.4: How does Wp-method's depth parameter affect HVLearn's performance and accuracy?

As discussed in Section IV-D, the number of queries performed by the Wp-method is exponential on the customizable depth parameter. We evaluated how this exponential term is

affecting the number of queries in practice and moreover, what is the effect of different values of the depth parameter on the correctness of the models inferred by HVLearn.

For our first experiment, we explore the correlation between the overall number of membership queries and the corresponding depth parameter. The results of this experiment are presented in Figure 6 and Table V. In order to ensure that the experiment finishes within a reasonable time, we further reduced the alphabet size only to two symbols. the results clearly show that the dependence between the depth parameter and the overall number of queries performed by the learning algorithm is clearly exponential, and in fact exactly matches the $O(|\Sigma|^d)$ bound where d is the depth parameter as discussed in Section IV-D. Notice that when the depth parameter of the Wp-method is set to a value less than 8, HVLearn fails to infer any aspect of the target implementation and outputs a single state DFA model that rejects all hostnames as shown in Table V.

> ***Result 4:*** *Large values of the Wp-method depth parameter result in impractical running times while small values result in incomplete models.*

RQ.5: How much improvement is offered by the equivalence query optimization in HVLearn?

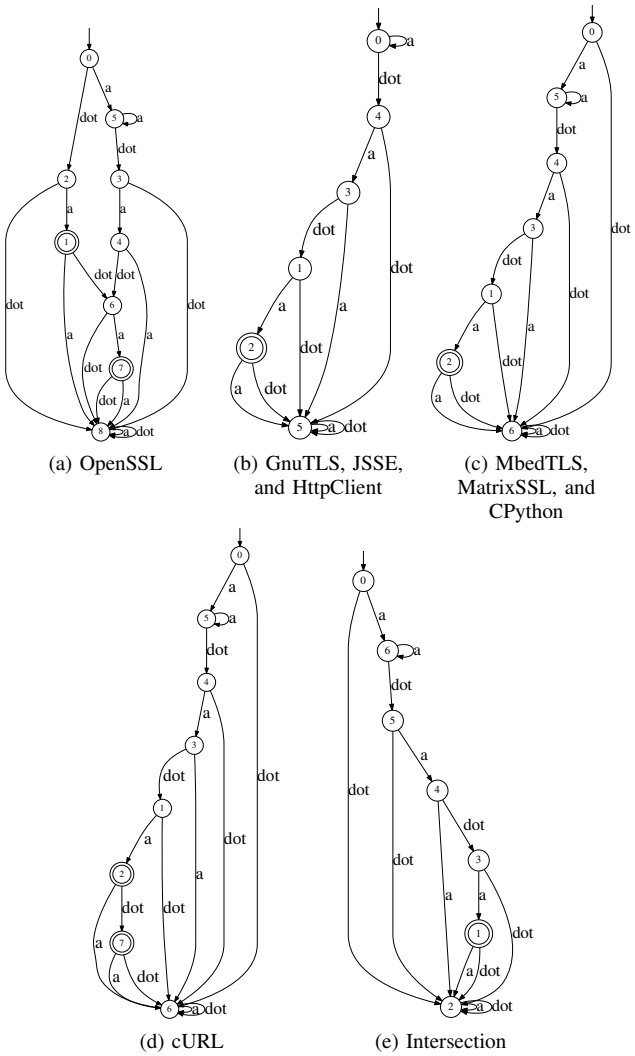
The previous experiment clearly demonstrates that the Wp-method alone is not efficient enough to accurately analyze a variety of different templates with HVLearn. Using our full alphabet, inferring a complete model for the common name "*.aaa.aaa" requires the depth parameter to be ≥ 8 as shown in Table V. With our full alphabet of 13 symbols this would require around 2^{30} queries based on the query complexity of the algorithm. We find that even running the algorithm with a depth of 6, which is still not able to infer a complete model, results in more than 68 million queries.

Therefore, our equivalence query optimization is a crucial component of HVLearn that allows it to produce accurate DFA models that can be used to evaluate the security and correctness of the implementations. As we can see from Table V, using our equivalence query optimization and a depth parameter of just 1, our system is able to produce a complete model for a given certificate template. Running the same experiment with the alphabet size 15, we found that HVLearn infers a correct model using only 14,812 queries as shown in Table IV.

> ***Result 5:*** *EQ optimization is providing, in some cases, over one order of magnitude improvement on the number of queries required to infer a complete DFA model.*

TABLE V
THE NUMBER OF QUERIES NEEDED TO LEARN THE DFA MODEL OF CPYTHON CERTIFICATE VERIFICATION FOR DIFFERENT WP-METHOD DEPTH VALUES

Wp. Depth	W/o EQ Optimization			With EQ Optimization		
	#Queries	#States	Complete?	#Queries	#States	Complete?
1	7	1	✗	226	11	✓
2	15	1	✗	448	11	✓
3	31	1	✗	890	11	✓
4	63	1	✗	1,778	11	✓
5	127	1	✗	3,554	11	✓
6	255	1	✗	7,104	11	✓
7	511	1	✗	14,207	11	✓
8	28,415	11	✓	28,415	11	✓
9	56,831	11	✓	56,831	11	✓
10	113,663	11	✓	113,663	11	✓



(a) OpenSSL (b) GnuTLS, JSSE, and HttpClient (c) MbedTLS, MatrixSSL, and CPython

(d) cURL (e) Intersection

Fig. 7. SSL/TLS implementations' DFA and intersection DFA with CN/DNS: `*.a.a` and alphabet: {`a`, `.`}

F. Specification Extraction

Let us now examine how we can utilize HVLearn's specification extraction functionality in order to infer a practical specification for the rule corresponding to the common name "*.a.a". This rule corresponds to the basic wildcard certificate case where a wildcard is found in the leftmost label of the identifier. Nevertheless, Figure 7 demonstrates that even for this simple rule, the corresponding DFA models for different implementations present obvious discrepancies. For example, DFA model (a) accepts the hostname ".a", model (b) accepts the hostname ".a.a", while model (d) accepts the hostname "a.a.a.". Only model (c) perform the most intuitive matching

by only accepting hostnames matching the regular expression "a+.a.a" (here '+' denotes one or more repetitions of the character 'a').

By computing the intersection between all DFA models, we obtain the intersection DFA model (e). Our first observation is that the intersection DFA has only 6 states and it is thus very compact as discussed in Section V-F. Furthermore, we notice that the intersection DFA is the same as DFA (c) that corresponds to the most natural implementation of the corresponding rule. More importantly, even if we compute the intersection without including model (c), we will still infer the same specification. Thus, we conclude that computing the intersection of DFA models, even from implementations which fail in different ways, can often produce compact and natural specifications.

Size of inferred models. In general, the actual size of the inferred models is heavily dependent on the implementation details of the tested system. However, we expect that the DFA models inferred by our system will have around $l + 2$ states, where l is the length of the common name in the certificate template. Indeed, if we consider the inferred DFAs in Figure 7 we can notice that, for the common name "*.a.a" with length $l = 5$, the average number of states is 6.9, which is very close to the expected 7 states. Intuitively, the reasoning behind this size is that a DFA for matching a string of length l is expected to have $l + 2$ states in general where l states are moving the DFA forward towards the accepting state while the additional 2 states include the initial state and a *sink* state where the DFA goes when no match is found.

VI. Case Study of Bugs

The goal of our study aims at understanding the severity of potential exploitation by incorrect or unclear hostname check in certificate verification. We are also interested in finding any inconsistency of SSL/TLS implementations' hostname checks with what RFC specifies. In this section, we present some interesting cases we achieved from the result of our experiment or corner cases we found.

A. Wildcards within A-labels in IDN identifiers

RFC 6125 strictly prohibits matching a certificate with an identifier containing wildcards embedded within an A-label of an IDN. For a certificate with an identifier of the form "xn−−aa*", it is very difficult to predict the set of unicode strings that will be matched after they are transformed into the punycode format due to the complexity of the transformation process. This inability to easily predict the set of hostnames which match an A-label with an embedded wildcard often present avenues for man-in-the-middle attacks.

Hostname verification implementations which match identifiers with wildcards embedded within A-labels have been found recently in the Ruby OpenSSL extension [28] and the NSS library used by Mozilla Firefox [27]. These issues were identified as security vulnerabilities by the developers of the corresponding products.

Using HVLearn, we identified that both JSSE and HttpClient (without using `PublicSuffixMatcher` in constructor) were also vulnerable to this issue. Our tool also reported that the other tested libraries/applications were not affected.

B. Confusing order of checking between CN and SAN identifiers.

RFC 6125 explicitly specifies that applications should not attempt to match the hostname with the subject CN when any subjectAltName identifiers are present, regardless of whether there is a match in subjectAltName as shown in Section II). We found a number of violations of that rule using HVLearn as described in Table II. We also found that MatrixSSL exhibits an interesting behavior in such cases.

More specifically, MatrixSSL matches the CN identifier before attempting to match any identifiers in the SAN even if they are present in the certificate. Note here that the CN does not have any strong restrictions on its content and may even contain non-FQDN characters (e.g., UTF-8).

Therefore, it is possible that certain certificate authorities, following the instructions in RFC 6125, will not check the CN in the presence of SAN identifiers and will issue a certificate regardless of the value in the CN as long as the user is successfully identified as the owner of the domains in the SAN identifier. Albeit natural, this choice will render applications using MatrixSSL vulnerable to a simple man-in-the-middle attack.

Specifically, an attacker can generate a signed certificate with a SAN identifier for a domain owned by the attacker, say "www.attacker.com" and have the CN field set to the victim domain, say "www.bank.com". MatrixSSL will first check the CN and omit to check the SAN identifiers. Therefore, MatrixSSL will allow the attacker to hijack any domain which is present in the CN field (e.g., www.bank.com).

C. Hijacking IP-based certificates

Section 2.3.1 of domain names implementation and specification in RFC [16] dictates that the preferred name (label) should only begin with a letter character. However, RFC [17] changed this restriction to allow the first character to be a letter or a digit. This change introduced valid DNS names which are identical to IP addresses.

Unfortunately, the fact that IP addresses are also valid DNS names may open a new avenue for an attack as we describe below. Notice that, for this attack to become practical, a numeric Top Level Domain (TLD) in the range 0-255 must exist, something that is currently unavailable. Nevertheless, our description should be taken as a precautionary note for new TLDs.

The attack is based on the fact that certain implementations first check if the given hostname matches the certificate's CN/SAN as a domain name and afterward as IP address. Therefore, consider an attacker controlling an IP address, say 80.50.12.33 and holding an IP-based certificate with that IP address. Then, assuming that "33" is a valid TLD, the same entity is automatically in possession of a certificate for

TABLE VI
BEHAVIORS OF SSL/TLS IMPLEMENTATIONS FOR X.509 CERTIFICATES WITH IPV4 ADDRESSES IN CN/SUBJECTALTNAME

SSL/TLS Libs/Apps	Certificate with IPv4 in	
	Subject CN	SubjectAltName DNS
OpenSSL	app	app
GnuTLS	accept	accept
MbedTLS	accept*	accept*
MatrixSSL	accept	accept
JSSE	reject	reject
CPython SSL	accept	reject
HttpClient	accept	reject
cURL	accept	reject

app: library lets application choose the identifier type.
accept*: library/application does not support IP-based certification verification but allows IPv4-format string in hostname verification.

the *domain name* "80.50.12.33" and can perform man-in-the-middle attacks on that domain!

We evaluated whether this attack is feasible in current SSL/TLS implementations. Table VI shows the results of our evaluation. All libraries/applications which are marked with an *accept* either in the subject CN or subjectAltName DNS columns are vulnerable to this attack. Even though this issue is not currently exploitable, it presents a security risk for these libraries in case numerical TLDs are introduced in future.

D. Embedded NULL bytes in CN/SAN identifiers

In 2008, Kaminsky et al. [53] demonstrated a vulnerability in the hostname verification implementations of popular SSL/TLS libraries where early NULL-byte (\0) terminations in an X.509 CN causes some libraries to recognize different CN values. In a nutshell, a client accepts certificate from an attacker's subdomain "www.bank.com\0.attacker.com" when attempting to connect to "www.bank.com" and therefore allow the attacker to hijack the connection.

In order to defend against this attack, two lines of defense were followed. The first option was to reject any certificate containing NULL bytes embedded within any CN/SAN identifiers. The second line was to simply patch the API functions which retrieve the CN/SAN identifiers from the certificate in order to recover the entire identifier even in the presence of embedded NULL bytes.

We thoroughly evaluated the defense implemented in each SSL/TLS library. Table VII presents the results of our evaluation. The second column describes whether the SSL/TLS library allows embedded NULL bytes, the third column presents the corresponding API function which is used to retrieve the CN/SAN identifier, and the fourth column describes whether the API call also returns the length of the corresponding CN/SAN identifier. Note that this is a very important feature since, otherwise, the application using the SSL/TLS library cannot know where the identifier string is terminating. We notice that this important feature is implemented by all libraries except JSSE. Notice though that, even though JSSE is not returning the length of the corresponding identifier, since JSSE is written in Java, it is not vulnerable to the embedded NULL byte attacks because Java strings are not NULL terminated.

TABLE VII
SUPPORT FOR EMBEDDED NULL CHARACTER IN CN/SUBJECTALTNAME IN DIFFERENT SSL/TLS LIBRARIES

SSL Libraries	ID	Allows Embedded NULL?	Function / Structure Name	Returns Length
OpenSSL	CN	✓	X509_NAME_get_text_by_NID()	✓
	CN	✓	X509_NAME_get_text_by_OBJ()	✓
	CN	✓	X509_NAME_get_index_by_NID()[1]	✓
	CN	✓	X509_NAME_get_index_by_OBJ()[1]	✓
	SAN	✓	X509_get_ext_d2i()[2]	✓
GnuTLS	CN	✓	gnutls_x509_crt_get_dn_by_oid()	✓
	SAN	✓	gnutls_x509_crt_get_subject_alt_name()	✓
MbedTLS	CN	✓	mbedtls_x509_name	✓
	SAN	✗	mbedtls_x509_sequence	✓
MatrixSSL	CN	✗	x509DNattributes_t	✗
	SAN	✗	x509GeneralName_t	✓
JSSE	CN	✓	getSubjectX500Principal()	✗
	SAN	✓	getSubjectAlternativeNames()	✗
CPython SSL	— Functionality not exposed to apps —			

[1] followed by X509_NAME_get_entry()
[2] followed by sk_GENERAL_NAME_value()

Despite the fact that SSL/TLS implementations take precautions against embedded NULL byte attacks, this doesn't imply that the applications using the libraries are also secure. Indeed, applications implementing the hostname verification functionality must ensure that they do not use vulnerable functions such standard string comparison function from libc (e.g., `strcmp, strcasecmp, fnmatch`), as they match strings in NULL-termination style.

In order to evaluate the security of applications using SSL/TLS libraries against embedded NULL byte attacks, we conducted a manual audit against several applications. Unfortunately, we found several popular applications being vulnerable to man-in-the-middle attacks using embedded NULL byte certificates. Some examples include FreeRadius server [8] which is one of the most widely deployed RADIUS (Remote authentication dial-in user service) servers, OpenSIPS [12] which is a popular open-source SIP server, Proxytunnel [13] which is a stealth tunneling proxy, and Telex Anticensorship system [15] which is an open-source censorship-circumventing software.

An important takeaway from this section is that embedded NULL byte attacks, even though addressed at the SSL/TLS library level, still present a very realistic and overlooked threat for applications using these libraries.

VII. RELATED WORK

A. Securing SSL/TLS Implementations

The security analysis of different components of SSL/TLS implementations has been examined in a large number of projects. We provide a summary of the most related projects below. The key difference between these projects and ours is that none of these projects focused on automatically analyzing the correctness of the hostname verification part of SSL/TLS certificate validation implementations. Prior works didn't cover analyzing hostname verification in detail primarily

due to the hardness of accurately modeling the implementations. In this paper, we solve this problem by using automata learning techniques and demonstrating that they can accurately and efficiently infer DFA models of hostname verification implementations in a black-box manner.

Automated Analysis of SSL/TLS implementations. Brubaker et al. [36] and subsequently Chen et al. [39] used mutation-based differential testing to find certificate validation issues. However, in their case, the hostname verification functionality of the libraries under test is disabled in order to discover other certificate validation issues and thus, they cannot uncover bugs discovered by our work. He et al. [52] used static analysis to detect incorrect usage of SSL/TLS libraries APIs. Somorovsky [61] created TLS-Attacker a tool to fuzz the TLS implementations systematically. However, TLS-Attacker focused on finding bugs in the protocol level and did not analyze the hostname verification functionalities of SSL/TLS implementations. Finally, de Ruiter and Poll [41] used automata learning algorithms to infer models of the TLS protocol and manually inspected the machines to find bugs. Contrary to our approach, where we focus on analyzing hostname verification implementations, their work focused on the TLS state machine induced by the different messages exchanged during the TLS handshake.

Certificate validation. Georgiev et al. [50] studied different ways that SSL/TLS API was abused in non-browser software. They manually identified pervasive incorrect certificate validation in different SSL/TLS implementations on which critical software rely. Fahl et al. [45] investigated the incorrect usage of SSL/TLS API in Android apps. However, unlike HVLearn, none of these projects looked into the implementations of the API functions.

Parsing X.509 certificates with embedded NULL character. Kaminsky et al. [53] demonstrated that several hostname verification implementations mishandled embedded NULL characters in X.509 certificates and can be used to trick a CA into issuing a valid leaf certificate with the wrong subject name. However, they found this issue manually and did not have any automated techniques for analyzing hostname verification implementations. Moreover, these issues were supposed to be fixed by the SSL/TLS implementations but we find that several applications using incorrect APIs for extracting the identifier strings from a certificate still suffer from these vulnerabilities as described in Section VI.

Cryptographic attacks and implementation bugs. There is a large body of work on various cryptographic attacks on the SSL/TLS protocol implementations. The interested reader may consult [40] for a survey. These attacks include various protocol based attacks [35], [43], [44], [46] as well as timing attacks [37] and flaws in pseudo-random number generators [57]. Besides cryptographic attacks, implementation bugs may cause severe security vulnerabilities as demonstrated by recently discovered attacks [26], [56].

B. Automata inference and applications

Angluin [31] invented the L^* algorithm for learning deterministic finite automata (DFA) from membership and equivalence queries. In the following years, many variations and optimizations were developed, including the Kearns-Vazirani algorithm used in HVLearn [54]. The interested reader can read the paper by Balcazzar et al. [34] for a unified presentation of popular algorithms. Automata learning algorithms have been applied to infer models for various protocols such as EMV bank cards [29], electronic passports [30], TLS protocols [41] and TCP/IP implementations [47], [48].

Argyros et al. [33] utilized symbolic finite automata learning algorithms to create a differential testing framework and leveraged it to discover bugs in Web application firewalls. While our approach is similar in nature, we counter the problem of large alphabets by using only the necessary symbols for our analysis. Moreover, instead of using differential testing to simulate equivalence queries, our approach uses an optimized version of the Wp-method, which offers stronger correctness guarantees.

VIII. Conclusion

We designed, implemented and extensively evaluated HVLearn, an automated black-box automata learning framework for analyzing different hostname verification implementations. HVLearn supports automated extraction of DFA models from multiple different implementations as well as efficient differential testing of the inferred DFA models. Our extensive evaluation on a broad spectrum of hostname verification implementations found 8 RFC violations with serious security implications. Several of these RFC violations could enable active man-in-the-middle attacks. We also discovered 121 unique differences on average between each pair of inferred DFA models. In addition, given that the RFC specifications are often ambiguous about corner cases, we expect that the models inferred by HVLearn will be very useful to the developers for checking their hostname verification implementations against the RFC specifications and therefore can help in reducing the chances of undetected security flaws. We have made HVLearn open-source so that the community can continue to build on it. The framework can be accessed at https://github.com/HVLearn.

IX. Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This work was supported by the NSF under grants CNS-13-18415 and CNS-16-17670. Author Suphannee Sivakorn is also partially supported by the Ministry of Science and Technology of the Royal Thai Government. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or the NSF.

References

[1] https://gitlab.com/gnutls/gnutls/merge_requests/314.
[2] https://gitlab.com/gnutls/gnutls/issues/185.

[3] https://gitlab.com/gnutls/gnutls/issues/187.
[4] http://www.matrixssl.org/blog/releases/matrixssl_3_9_0.
[5] https://issues.apache.org/jira/browse/HTTPCLIENT-1802.
[6] American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/.
[7] cURL - Compare SSL Libraries. https://curl.haxx.se/docs/ssl-compared.html.
[8] FreeRADIUS. http://freeradius.org/.
[9] GnuTLS 3.5.10: X509 certificate API. https://goo.gl/ZSbNGb.
[10] Java Native Interface (JNI). https://docs.oracle.com/javase/8/docs/technotes/guides/jni/.
[11] libFuzzer - A Library for Coverage-guided Fuzz Testing. http://llvm.org/docs/LibFuzzer.html.
[12] OpenSIPS. https://github.com/OpenSIPS/opensips.
[13] proxytunnel. http://proxytunnel.sf.net.
[14] SLOCCount. https://www.dwheeler.com/sloccount/.
[15] Telex Anticensorship. https://github.com/ewust/telex.
[16] RFC 1035 - DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. https://tools.ietf.org/html/rfc1035, November 1987.
[17] RFC 1123 - Requirements for Internet Hosts – Application and Support. https://tools.ietf.org/html/rfc1123, October 1989.
[18] RFC 2818 - HTTP Over TLS. https://tools.ietf.org/search/rfc2818, May 2000.
[19] RFC 3492 - Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA). https://tools.ietf.org/html/rfc3492, March 2003.
[20] RFC 4985 - Internet X.509 Public Key Infrastructure Subject Alternative Name for Expression of Service Name. https://tools.ietf.org/html/rfc4985, August 2007.
[21] RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. https://tools.ietf.org/html/rfc5280, May 2008.
[22] RFC 5321 - Simple Mail Transfer Protocol. https://tools.ietf.org/html/rfc5321, October 2008.
[23] RFC 5890 - Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework. https://tools.ietf.org/html/rfc5890, August 2010.
[24] RFC 6125 - 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). https://tools.ietf.org/html/rfc6125, March 2011.
[25] RFC 6818 - Updates to the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. https://tools.ietf.org/html/rfc6818, January 2013.
[26] CVE-2014-0092, March 2014.
[27] CVE-2014-1492, March 2014.
[28] CVE-2015-1855, March 2015.
[29] F. Aarts, J. D. Ruiter, and E. Poll. Formal Models of Bank Cards for Free. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*, pages 461–468, 2013.
[30] F. Aarts, J. Schmaltz, and F. Vaandrager. Inference and Abstraction of the Biometric Passport. In *Proceedings of the International Conference on Leveraging Applications of Formal Methods, Verification, and Validation*, pages 673–686, 2010.
[31] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
[32] Apache Software Foundation. Apache HttpComponents - HttpComponents HttpClient Overview. https://hc.apache.org/httpcomponents-client-ga/.
[33] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias. SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 1690–1701, 2016.
[34] J. L. Balcázar, J. Díaz, R. Gavalda, and O. Watanabe. *Algorithms for Learning Finite Automata from Queries: A Unified View*, pages 53–72. Springer, 1997.
[35] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, pages 1–12, 1998.
[36] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, pages 114–129, 2014.
[37] D. Brumley and D. Boneh. Remote Timing Attacks Are Practical. In *Proceedings of the USENIX Conference on Security Symposium*, pages 1–1, 2003.
[38] Y. Chen and Z. Su. Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 793–804, 2015.
[39] Y. Chen and Z. Su. Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 793–804, 2015.
[40] J. Clark and P. C. van Oorschot. SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 511–525, 2013.
[41] J. De Ruiter and E. Poll. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the USENIX Conference on Security Symposium*, pages 193–206, 2015.
[42] Docjar. HostnameChecker. http://www.docjar.com/docs/api/sun/security/util/HostnameChecker.html.
[43] T. Duong and J. Rizzo. Here Come The $\oplus$ Ninjas. 2011.
[44] T. Duong and J. Rizzo. The CRIME Attack. 2012.
[45] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 50–61, 2012.
[46] N. J. A. Fardan and K. G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 526–540, 2013.
[47] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Learning Fragments of the TCP Network Protocol. In *Proceedings of the International Conference on Formal Methods for Industrial Critical Systems*, pages 78–93, 2014.
[48] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Proceedings of the International Conference on Computer Aided Verification*, pages 454–471, 2016.
[49] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test Selection Based on Finite State Models. *IEEE Transactions on software engineering*, 17(6):591–603, 1991.
[50] 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 ACM SIGSAC Conference on Computer and Communications Security*, pages 38–49, 2012.
[51] GNU Compilers. Gcov - Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Gcov.html.
[52] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang. Vetting SSL usage in applications with SSLint. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 519–534, 2015.
[53] D. Kaminsky, M. L. Patterson, and L. Sassaman. PKI Layer Cake: New Collision Attacks Against the Global x.509 Infrastructure. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, pages 289–303, 2010.
[54] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.
[55] D. Kozen. Lower Bounds for Natural Proof Systems. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, pages 254–266, 1977.
[56] A. Langley. Apple's SSL/TLS Bug. https://goo.gl/DzRLNq, 2014.
[57] A. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, Whit is right. *International Association for Cryptologic Research*, 2012.
[58] Oracle. Java Cryptography Architecture Oracle Providers Documentation. https://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html.
[59] H. Raffelt, B. Steffen, and T. Berg. LearnLib: A Library for Automata Learning and Experimentation. In *Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems*, pages 62–71, 2005.
[60] M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology Boston, 2006.
[61] J. Somorovsky. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504, 2016.

X. APPENDIX

A. Details of test hostname verification implementations

OpenSSL. has separate checking functions for each type identifiers as shown in Table I. In our testing, we use the default setup that supports matching wildcards. OpenSSL also provides support for applications to turn some of these hostname verification functions on or off by calling different setup functions (e.g., `X509_VERIFY_PARAM_set1_host` and `X509_VERIFY_PARAM_set1_email`).

GnuTLS. The GnuTLS check hostname function is designed for certificate verification for HTTPS supporting domain names, IPv4, and IPv6. Like OpenSSL, GnuTLS also provides the application to select whether to verify hostname with wildcard or not. By default, GnuTLS wildcard matching is enabled. We use the default setting for our experiments.

MbedTLS. The hostname verification functions in MbedTLS only supports checking for domain name verification.

MatrixSSL. A single function `matrixValidateCerts` is responsible for checking all different types of identifiers (e.g., DNS, IPv4, and email). The library does not include support for IPv6 yet. MatrixSSL also provides a separate function, `psX509ValidateGeneralName` that should be used before calling matrixValidateCerts for name checking for filtering out invalid input.

JSSE (Java Secure Socket Extension). SunJSSE [58], as part of the JSSE release, has internal built-in hostname checking support (sun.security.util.HostnameChecker [42]). It supports domain name, IPv4, and IPv6 verification through the HostnameChecker.match interface.

CPython SSL. CPython is the oldest and one of the most popular Python VM implementation. CPython's inbuilt SSL support depends on the OpenSSL library, but does not use OpenSSL's hostname verification function. Instead, it includes its own hostname verification implementation, `match_hostname` function. Currently, it only supports domain name and IP address verification but does not support email verification.

HttpClient. (Apache HttpClient) is used extensively in Web-services middleware such as Apache Axis 2 It supports IPv4, IPv6, and domain name verification [32]. By default the library provides a verify function in `DefaultHostnameVerifier` to perform the identity verification. The verifier can also be used with `PublicSuffixMatcher` object to perform additional checks.

cURL. By default, it uses OpenSSL [7] but implements its own hostname verification function `verifyhost` that supports domain name, IPv4, and IPv6 verification.

B. Developer Responses

We notified the developers of each affected library/application for all of our findings, including RFC violations and discrepancies. In this section, we present an overview of the developer responses for each different library/application.

GnuTLS. The GnuTLS team is currently working on a patch to fix the issue of seeking a match in the CN when an IP address identifier is in the subjectAltName [1]. The developers also plan to provide a way to specify the identifier type in order to avoid the confusion between hostnames and IP addresses [2]. Additionally, the team plans to remove a fallback option which matches an IP address with a subjectAltName DNS [9], thus resolving the potential attack presented in Section VI-C [3]. Finally, GnuTLS has recently introduced IDNA2008 support in version 3.5.9 and performs extensive checks to verify the format of the DNS names stored in the certificate.

MbedTLS. We are currently discussing the discovered issues with the MbedTLS team.

MatrixSSL. MatrixSSL is prioritizing the fixes for the RFC violations, including the incorrect order of checking between subject CN and subjectAltName identifier (violation of RFC 6125) and matching the local-part of an email address in a case-insensitive manner (violation of RFC 5280). These fixes are deployed in their new version 3.9.0 [4]. This version also addresses other discrepancies we reported by providing an optional flag for hostname input validation, and providing parameters for users in order to specify the type of the identifier (e.g., DNS, IP ADDR) in order to address the attack discussed in Section VI-C.

JSSE. The JSSE team does not consider RFC 6125 compliance to be a feature of the current version of the library. However, the team informed us that they are currently working on plans to add compliance with RFC 6125 in the next versions of the library.

CPython SSL. CPython plans to deprecate their hostname verification implementation and directly use OpenSSL's implementation in the next release.

OpenSSL. The OpenSSL team decides not to address the issue of matching a partial hostname suffix of a subject CN/subjectAltName, as this discrepancy is not an RFC violation. For the other discrepancies e.g., matching a wildcard in a public suffix or matching an invalid hostname, the OpenSSL team believes that they should be handled at the application level or by certificate authorities and therefore, they should not be fixed in the library itself.

HttpClient. The HttpClient team has addressed the violations of matching a subject CN in case sensitive manner (violation of RFC 6125 and RFC 5280) and attempting to match subject CN when a subjectAltName is present (violation of RFC 6125). These issues are resolved in version 4.5.3, which is currently an alpha release [5]. The HttpClient team decided not to address the other reported issues as they are handled correctly if the application calls the `DefaultHostnameVerifier` with the `PublicSuffixMatcher` in the verifier constructor.

C. Detailed list of discrepancies

In Table VIII, we present a detailed list of the discrepancies discovered between various SSL/TLS libraries and applications.

TABLE VIII
SAMPLE STRINGS ACCEPTED BY THE AUTOMATA INFERRED FROM DIFFERENT HOSTNAME VERIFICATION IMPLEMENTATIONS

Test Certificate Identifier Template	OpenSSL	GnuTLS	MbedTLS	MatrixSSL	JSSE	CPython SSL	HttpClient	cURL
Wildcard in Certificate								
*.aaa.aaa	a.aaa.aaa .aaa.aaa *.aaa.aaa .aaa a.aaa.aaa\0 .aaa.aaa\0 .aaa\0 *.aaa.aaa\0	.aaa.aaa .aaa.aaa\0	a.aaa.aaa a.aaa.aaa\0	a.aaa.aaa a.aaa.aaa\0	a.aaa.aaa	a.aaa.aaa	.aaa.aaa	a.aaa.aaa a.aaa.aaa.\0 a.aaa.aaa\0 a.aaa.aaa.
aaa.*.aaa	aaa.*.aaa .aaa .*.aaa aaa.*.aaa\0 .aaa\0 .*.aaa\0	aaa.*.aaa aaa.*.aaa\0	aaa.*.aaa aaa.*.aaa\0	none	aaa.a.aaa	aaa.*.aaa	aaa..aaa	aaa.*.aaa aaa.*.aaa.\0 aaa.*.aaa\0 aaa.*.aaa.
a*.aaa.aaa	aa.aaa.aaa a.aaa.aaa a*.aaa.aaa .aaa.aaa .aaa aa.aaa.aaa\0 a.aaa.aaa\0 a*.aaa.aaa\0 .aaa.aaa\0 .aaa\0	a*.aaa.aaa a*.aaa.aaa\0	a*.aaa.aaa a*.aaa.aaa\0	none	a.aaa.aaa	a.aaa.aaa	a.aaa.aaa	aa.aaa.aaa aa.aaa.aaa.\0 aa.aaa.aaa\0 aa.aaa.aaa.
aaa.a*.aaa	aaa.a*.aaa .aaa .a*.aaa aaa.a*.aaa\0 .aaa\0 .a*.aaa\0	aaa.a*.aaa aaa.a*.aaa\0	aaa.a*.aaa aaa.a*.aaa\0	none	aaa.a.aaa	aaa.a*.aaa	aaa.a.aaa	aaa.a*.aaa aaa.a*.aaa.\0 aaa.a*.aaa\0 aaa.a*.aaa.
xn--aaa*.aaa	.aaa .aaa\0	xn--aaa*.aaa xn--aaa*.aaa\0	xn--aaa*.aaa xn--aaa*.aaa\0	none	xn--aaa.aaa	xn--aaa*.aaa	xn--aaa.aaa	xn--aaa*.aaa xn--aaa*.aaa.\0 xn--aaa*.aaa\0 xn--aaa*.aaa.
*.xn--aaa.aaa	a.xn--aaa.aaa .aaa .xn--aaa.aaa *.xn--aaa.aaa a.xn--aaa.aaa\0 .aaa\0 .xn--aaa.aaa\0 *.xn--aaa.aaa\0	.xn--aaa.aaa .xn--aaa.aaa\0	.xn--aaa.aaa .xn--aaa.aaa\0	none	a.xn--aaa.aaa	a.xn--aaa.aaa	.xn--aaa.aaa	a.xn--aaa.aaa a.xn--aaa.aaa.\0 a.xn--aaa.aaa\0 a.xn--aaa.aaa.
xn--aaa.*.aaa	.aaa .*.aaa xn--aaa.*.aaa .aaa\0 .*.aaa\0 xn--aaa.*.aaa\0	xn--aaa.*.aaa xn--aaa.*.aaa\0	xn--aaa.*.aaa xn--aaa.*.aaa\0	none	xn--aaa.a.aaa	xn--aaa.*.aaa	xn--aaa..aaa	xn--aaa.*.aaa xn--aaa.*.aaa.\0 xn--aaa.*.aaa\0 xn--aaa.*.aaa.
Wildcard Unclear Practices								
*.aaa	.aaa *.aaa .aaa\0 *.aaa\0	none	a.aaa a.aaa\0	a.aaa a.aaa\0	a.aaa	a.aaa	.aaa	*.aaa *.aaa.\0 *.aaa\0 *.aaa.
a*b*c*.aaa.aaa	a*b*c*.aaa.aaa .aaa.aaa .aaa a*b*c*.aaa.aaa\0 .aaa.aaa\0 .aaa\0	a*b*c*.aaa.aaa a*b*c*.aaa.aaa\0	a*b*c*.aaa.aaa a*b*c*.aaa.aaa\0	none	abc.aaa.aaa	none	ab*c*.aaa.aaa	aab*c*.aaa.aaa aab*c*.aaa.aaa.\0 aab*c*.aaa.aaa\0 aab*c*.aaa.aaa.
..aaa.aaa	.aaa.aaa .*.aaa.aaa *.*.aaa.aaa .aaa .aaa.aaa\0 .aaa\0 .*.aaa.aaa\0 *.*.aaa.aaa\0	.*.aaa.aaa .*.aaa.aaa\0	a.*.aaa.aaa a.*.aaa.aaa\0	none	a.a.aaa.aaa	a.*.aaa.aaa	.*.aaa.aaa	a.*.aaa.aaa a.*.aaa.aaa.\0 a.*.aaa.aaa\0 a.*.aaa.aaa.
*b.aaa.aaa	ab.aaa.aaa b.aaa.aaa .aaa.aaa *b.aaa.aaa .aaa ab.aaa.aaa\0 b.aaa.aaa\0 .aaa.aaa\0 .aaa\0 *b.aaa.aaa\0	b.aaa.aaa b.aaa.aaa\0	*b.aaa.aaa *b.aaa.aaa\0	none	ab.aaa.aaa b.aaa.aaa	b.aaa.aaa	b.aaa.aaa	ab.aaa.aaa ab.aaa.aaa.\0 ab.aaa.aaa\0 ab.aaa.aaa.
.aaa.aaa	.aaa.aaa .aaa .aaa.aaa\0 .aaa\0	none	.aaa.aaa .aaa.aaa\0	none	aaa.aaa	.aaa.aaa	.aaa.aaa	.aaa.aaa .aaa.aaa.\0 .aaa.aaa\0 .aaa.aaa.
Email Address								
SAN email: *@aaa.aaa	*@aaa.aaa *@aaa.aaa\0	*@aaa.aaa *@aaa.aaa\0	–	none	–	–	–	–
SAN email: aaa@*	aaa@* aaa@*\0	aaa@* aaa@*\0	–	none	–	–	–	–
SAN email: aaa@*.aaa	aaa@*.aaa aaa@*.aaa\0	aaa@*.aaa aaa@*.aaa\0	–	none	–	–	–	–
SAN email: aaa@aaa.*	aaa@aaa.* aaa@aaa.*\0	aaa@aaa.* aaa@aaa.*\0	–	none	–	–	–	–
SAN email: AAA@aaa.aaa	AAA@aaa.aaa AAA@aaa.aaa\0	AAA@aaa.aaa AAA@aaa.aaa\0	–	aaa@aaa.aaa aaa@aaa.aaa\0	–	–	–	–
SAN email: aaa@AAA.aaa	aaa@aaa.aaa aaa@aaa.aaa\0	aaa@aaa.aaa aaa@aaa.aaa\0	–	aaa@aaa.aaa aaa@aaa.aaa\0	–	–	–	–
IP Address								
SAN IP Addr: *.111.111.111	none	none	–	none	none	none	none	none