NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science

SSLINT: A Tool for Detecting TLS Certificate Validation Vulnerabilities

April 2016

Boyuan He[1], Vaibhav Rastogi[2], Yinzhi Cao[3], Yan Chen[4],
V.N. Venkatakrishnan[5], Chunlin Xiong[1], Runqing Yang[1], and Zhenrui Zhang[1]

[1]Zhejiang University [2]University of Wisconsin-Madison
[3]Leihigh University [4]Northwestern University [5]University of Illinois, Chicago
heboyuan@zju.edu.cn vrastogi@wisc.edu yinzhi.cao@lehigh.edu ychen@northwestern.edu
venkat@uic.edu chunlinxiong@zju.edu.cn rainkin1993@zju.edu.cn jerryzh@zju.edu.cn

Technical Report NU-EECS-16-07

Abstract

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols have become the security backbone of the Web and Internet today. Many systems including mobile and desktop applications are protected by SSL/TLS protocols against network attacks. However, many vulnerabilities caused by incorrect use of SSL/TLS APIs have been uncovered in recent years. Such vulnerabilities, many of which are caused due to poor API design and inexperience of application developers, often lead to confidential data leakage or man-in-the-middle attacks. In this paper, to guarantee code quality and logic correctness of SSL/TLS applications, we design and implement SSLINT, a scalable, automated, static analysis system for detecting incorrect use of SSL/TLS APIs. SSLINT is capable of performing automatic logic verification with high efficiency and good accuracy. To demonstrate it, we apply SSLINT to one of the most popular Linux distributions – Ubuntu. We find 29 previously unknown SSL/TLS vulnerabilities in Ubuntu applications, most of which are also distributed with other Linux distributions.

SSLINT: A Tool for Detecting TLS Certificate Validation Vulnerabilities

Boyuan He[1], Vaibhav Rastogi[2], Yinzhi Cao[3], Yan Chen[4],
V.N. Venkatakrishnan[5], Chunlin Xiong[1], Runqing Yang[1], and Zhenrui Zhang[1]

[1]Zhejiang University [2]University of Wisconsin-Madison
[3]Leihigh University [4]Northwestern University [5]University of Illinois, Chicago
heboyuan@zju.edu.cn vrastogi@wisc.edu yinzhi.cao@lehigh.edu ychen@northwestern.edu
venkat@uic.edu chunlinxiong@zju.edu.cn rainkin1993@zju.edu.cn jerryzh@zju.edu.cn

Abstract—Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols have become the security backbone of the Web and Internet today. Many systems including mobile and desktop applications are protected by SSL/TLS protocols against network attacks. However, many vulnerabilities caused by incorrect use of SSL/TLS APIs have been uncovered in recent years. Such vulnerabilities, many of which are caused due to poor API design and inexperience of application developers, often lead to confidential data leakage or man-in-the-middle attacks. In this paper, to guarantee code quality and logic correctness of SSL/TLS applications, we design and implement SSLINT, a scalable, automated, static analysis system for detecting incorrect use of SSL/TLS APIs. SSLINT is capable of performing automatic logic verification with high efficiency and good accuracy. To demonstrate it, we apply SSLINT to one of the most popular Linux distributions – Ubuntu. We find 29 previously unknown SSL/TLS vulnerabilities in Ubuntu applications, most of which are also distributed with other Linux distributions.

1 INTRODUCTION

Secure Socket Layer (SSL) and its successor Transport Layer Security (TLS) provide end-to-end communication security over the Internet. Based on the model of Public Key Infrastructure (PKI) and X509 certificates, SSL/TLS is designed to guarantee confidentiality, authenticity, and integrity for communications against Man-In-The-Middle (MITM) attacks.

The details of SSL/TLS protocol are complex, involving six major steps during the handshaking protocol [1]. To ease the burden of developers, these details are encapsulated inside open source SSL/TLS libraries such as OpenSSL, GnuTLS, and NSS (Network Security Services). However, recent work [2] has shown that incorrect use of such libraries could lead to certificate validation problems, making applications vulnerable to MITM attacks. Their work sheds light on a very important issue for Internet applications, and since then SSL implementations have received considerable scrutiny and follow-up research [3]–[8].

In this backdrop, we focus on the problem of large-scale detection of SSL certificate validation vulnerabilities in client software. By large-scale, we refer to techniques that could check, say, an entire OS distribution for the presence of such vulnerabilities. Previous research, including [2], on finding SSL vulnerabilities in client-server applications, mostly relied on a black-box testing approach. Such an approach is not suitable for large-scale vulnerability detection, as it involves activities such as installation, configuration and testing, some of which involve a human-in-the-loop.

In particular, we ask the following research question: *Is it possible to design scalable techniques that detect incorrect use of APIs in applications using SSL/TLS libraries?* This question poses the following challenges:

- ***Defining and representing correct use***. Given an SSL library, how do we model correct use of the API to facilitate detection?
- ***Analysis techniques for incorrect usage in software.*** Given a representation of correct usage, how do we design techniques for analyzing programs to detect incorrect use?
- ***Identifying candidate programs in a distribution.*** From an OS distribution, how do we identify and select candidate programs using SSL/TLS libraries?
- ***Precision, Accuracy and Efficiency.*** How do we design our techniques so that they offer acceptable results in terms of precision, accuracy and efficiency?

We address these questions in this paper proposing an approach and tool called SSLINT– a scalable, automated, static analysis tool – that is aimed towards automatically identifying incorrect use of SSL/TLS APIs in client-side applications.

The main enabling technology behind SSLINT is the use of graph mining for automated analysis. By representing both the correct API use and SSL/TLS applications as program dependence graphs (PDGs), SSLINT converts the problem of checking correct API use into a graph query problem. These representations allow for the correct use patterns to precisely capture temporal sequencing of API calls, data flows between arguments and returns of a procedure, data flows between vari-

ous program objects, and path constraints. Using these representations we develop rich models of correct API usage patterns, which are subsequently used by a graph matching procedure for vulnerability detection.

To evaluate SSLINT in practice, we applied it to the source code of 492 software packages from Ubuntu. The result shows that SSLINT discovers 29 previously unknown SSL/TLS vulnerabilities. Then, we reported our findings to all the developers of software with such vulnerabilities and received 14 confirmations – out of which, 5 have already fixed the vulnerability based on our reports. For those we have not received confirmations from, we validated them by performing MITM attacks, and the result shows that they are all vulnerable.

To summarize, this paper makes the following contributions:

- ***SSL/TLS library signature.*** We model the correct API usage as SSL/TLS library signatures based on PDGs.
- ***Graph query matching.*** SSLINT is able to perform automated, scalable graph queries to match SSL/TLS library signatures for all the SSL/TLS APIs, and report a vulnerability if the matching fails.
- ***Automated search of applications relying on SSL/TLS libraries.*** We leverage on existing package managers in Ubuntu for automatic compiling and analyzing, and then acquire all the target applications with SSL/TLS libraries as their building dependences.
- ***Preprocessing analysis to reduce program needed to be analyzed.*** We introduce a blacklist based preprocessing on low-accuracy, quickly-computed call graph of the program to prune code that may not lead to vulnerabilities.
- ***Analysis of libraries*** Some libraries pose significant challenge to static analysis as the data flows are completed only when they are linked with the client programs. In such cases, we have manually written client programs to enable accurate static analysis for libraries.
- ***Automated reachability analysis for vulnerable libraries.*** When our analysis finds vulnerabilities in library, we conduct an automatic reachability analysis starting from the library API to determine if the vulnerable code is reachable from public API functions and thus makes the client programs possibly vulnerable.
- ***Evaluation results.*** We discover 29 previously unknown SSL/TLS vulnerabilities in software packages from the Ubuntu 12.04 source. These vulnerabilities were confirmed through manual auditing (in some cases, requiring us to develop programs exercising the vulnerable code) or by the software developers themselves.

The remainder of this paper proceeds as follows: Section 2 provides relevant background in SSL/TLS and static analysis. Section 3 provides the motivation of the study in this paper as well as the detailed discussion of the techniques incorporated into SSLINT. Section 4 discusses the implementation of SSLINT. Section 5 and 6 give the evaluation results of SSLINT in Ubuntu software packages and discusses the accuracy and limitations. Section 7 presents related work and Section 8 concludes the paper.

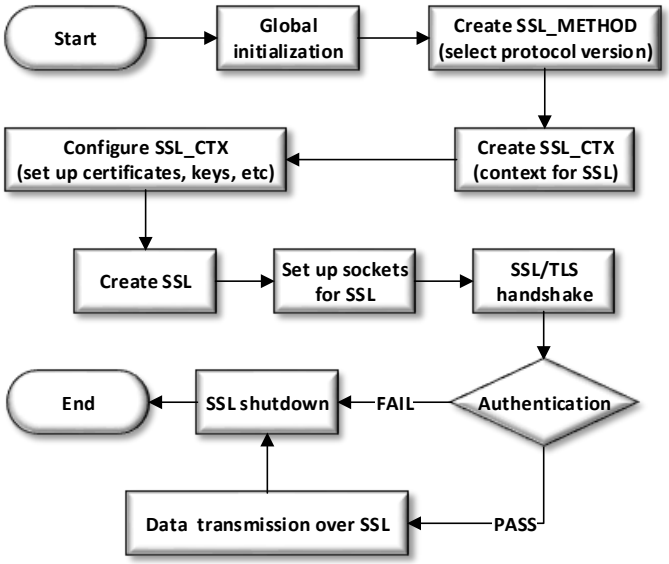

Fig. 1. Overview of SSL application with OpenSSL APIs.

2 OVERVIEW

2.1 Overview of SSL/TLS

SSL/TLS provides end-to-end communication security including confidentiality, message integrity, and site authentication between a client and a server, even if the network between the client and the server is under control of an adversary. The client verifies the authenticity of the server by validating an X.509 certificate chain from the server.

Listing 1. Certificate chain validation with OpenSSL APIs.

```
1  const SSL_METHOD *method;
2  SSL_CTX *ctx;
3  SSL *ssl;
4  [...]
5  //select protocol
6  method = TLSv1_client_method();
7  [...]
8  //Create CTX
9  ctx = SSL_CTX_new(method);
10 [...]
11 //Create SSL
12 ssl = SSL_new(ctx);
13 [...]
14 //set SSL_VERIFY_PEER flag to enforce
       certificate chain validation during
       handshake
15 SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER,...);
16 [...]
17 //Start handshake
18 SSL_connect(ssl);
19 [...]
```

SSL/TLS libraries encapsulate the core functionality of the SSL/TSL protocols, and export an API that allows a client application to setup and validate SSL connections. For validation in particular, the client needs to validate the authenticity of each certificate issued by certificate authority (CA) in the chain, and we now present the validation process that checks for the following properties:

P1. Hostname validity. A client needs to validate that the first certificate is issued for the target server. In particular, the client checks the `CommonName (CN)` attribute in the `Subject` field of an X.509 certificate, which contains the hostname of the certificate holder. We refer this checking step as `hostname validation` for the rest of the paper.

P2. Certificate chain validity. In a certificate chain, a client needs to validate that each certificate is issued by the CA of its parent certificate or the root CA, and the CA is authorized to issue certificates. In particular, the client checks whether the `issuer` field of the certificate matches the CA of its parent certificate or the root CA, and whether the `CA` attribute of `basicConstraint` field of its parent certificate is true. In addition, a client needs to validate whether each certificate in the chain expires, i.e., check the `validity` field of each certificate. Together, we refer the certificate chain validation and expiration date validation steps as *certificate validation* for the rest of the paper.

2.2 A typical SSL application

Let us consider an example of how a typical application that uses an SSL/TLS library is implemented. Figure 1 is an overview of an SSL/TLS application using OpenSSL APIs. The application first initializes variables, and creates a new "context" with both local certificates and keys. Then, the application establishes a connection with the server through an SSL handshake [1], in which the certificate chain is validated. If successful, the client and the server exchange data through the established connection in a secure fashion.

While Figure 1 is illustrative of how a typical application uses OpenSSL, it is worth noting that OpenSSL provides more than one API combination of implementing the connection setup, validation and shutdown. Such rich API surface allows the developer considerable latitude in creating an SSL/TLS connection. For instance, let us consider two code examples of applications that use the OpenSSL API to perform validation in Listing 1 and Listing 2 respectively. The code in Listing 1 performs validation during the handshake step and drops connection if the validation fails. In comparison, the code in Listing 2 validates a server's certificate *after* a successful establishment of an SSL/TLS connection. Both API uses are acceptable, provided that the certificate validation is correct.

Listing 2. Certificate chain validation with OpenSSL APIs.

```
1  const SSL_METHOD *method;
2  SSL_CTX *ctx;
3  SSL *ssl;
4  X509 *cert = NULL;
5  [...]
6  //select protocol
7  method = TLSv1_client_method();
8  [...]
9  //Create CTX
10 ctx = SSL_CTX_new(method);
11 [...]
12 //Create SSL
13 ssl = SSL_new(ctx);
14 [...]
15 //Start handshake
16 SSL_connect(ssl);
17 [...]
18 cert = SSL_get_peer_certificate(ssl);
19 if (cert != NULL){
20     if(SSL_get_verify_result(ssl)==X509_V_OK)
           {
21         //The validation succeeds.
22     }
23     else{
24         //The validation fails and the
               connection terminates.
25     }
26 }
27 else{
28     //The validation fails and the connection
           terminates.
29 }
30 [...]
```

2.3 Vulnerable SSL application

Ideally, SSL libraries should implement all the aforementioned validation functionalities, i.e., perform built-in certificate validation and provide APIs for application interactions. However, as documented in recent work [2], these SSL/TLS library APIs are poorly designed and require careful use by the programmer to get right. Most often, programmers do not supply that level of attention, and this leads to vulnerabilities in applications that use them. We discuss two types of vulnerabilities here, corresponding to a violation of either P1 or P2 discussed above. For illustration purpose, we provide a vulnerable code example that we found in Scrollz IRC Client [9] in Listing 3. (See Section 5 for details.) Note that Scrollz IRC Client uses GnuTLS, a different SSL/TLS library.

In Listing 3, both hostname and certificate validations are missing, so one can perform MITM attacks exploiting either of the two to compromise users of the IRC client. Note that GnuTLS does provide APIs for both validations, but the developers fail to use such APIs and perform the validations.

V1. Hostname validation vulnerability. Hostname validation vulnerability is because a client does not validate the hostname of the first certificate in the chain, in violation of the property P1. The correct validation is as follows. The client first reads the entirely certificate chain by `gnutls_certificate_get_peers`. Then, the client chooses the first certificate in the chain by `gnutls_x509_crt_import`

and validates the hostname in the certificate by `gnutls_x509_crt_check_hostname`. Finally, the client checks the return value of `gnutls_x509_crt_check_hostname` to see whether the validation is successful. Scrollz fails to validate hostname as shown in Listing 3.

To launch an MITM attack exploiting this vulnerability, an attacker needs to first use Domain Name Server (DNS) poisoning. Then, the connection request from a client to a server with a poisoned hostname is now forwarded to the attacker. The attacker can supply the client with a valid certificate issued to the attacker's domain name. Because the client application (Scrollz IRC Client) does not check the hostname of the certificate, it accepts the vulnerable connection, and subsequently exposes data in the connection to the attacker.

Listing 3. Vulnerable Code from Scrollz IRC Client.

```
1  gnutls_init(&server_list[server].session,
       GNUTLS_CLIENT);
2  [...]
3  gnutls_credentials_set(server_list[server].
       session, GNUTLS_CRD_CERTIFICATE,
       server_list[server].xcred);
4  [...]
5  err = gnutls_handshake(server_list[server].
       session);
6  [...]
```

V2. Certificate validation vulnerability. Certificate validation vulnerability is because a client does not check issuers of the certificates in the certificate chain. The correct validation is as follows. The client calls `guntls_certificate_verify_peer2` for certificate validation, checks the return value, and compares the `status` flag with multiple constant representing different errors. Similarly, Scrollz fails to validate certificate as shown in Listing 3.

To launch an MITM attack exploiting this vulnerability, an attacker can replace the original certificate of the server with a self-signed certificate. Because the self-signed certificate appears to be valid to the client, the client accepts the connection with the attacker. Later on, when the client communicates with the attacker using the self-signed certificate, the attacker sniffs the traffic and forwards the traffic to the original server so that the client still functions correctly.

In summary, a client should not send or receive any application data until it confirms the server's identity by certificate and hostname validations. In practice, programmers may forget those two validations and write vulnerable client software.

2.4 Discussion

Our goal is to perform large-scale, vulnerability detection of hostname and certificate validation vulnerabilities in applications that use SSL/TLS libraries. By large-scale, we mean that the detection needs to work at the level of an OS distribution (that contains hundreds of software programs) to look for vulnerabilities in all its deployed software. Prior work [2] in this area relied on manual analysis and black-box fuzzing. While this has yielded impressive results, the methodology adopted there is unsuitable for large-scale vulnerability analysis.

One approach to look for vulnerabilities is to perform automated testing of applications that use SSL/TLS libraries. This might entail automated installation and deployment and testing of the client with a corresponding SSL/TLS-enabled server. While this might initially seem easy, automation of this kind is actually hard. Consider a mail-client that we would like to test using this approach. This mail-client needs to be set-up, configured to use a particular mail-server, and the corresponding server-side needs to be configured and deployed. While none of these tasks pose serious technical challenges, automating them is both tedious as well as unscalable.

An alternative option is to use a static analysis approach. In this, we can look for whether the code of the application follows some safe conventions for SSL/TLS software development that avoids the vulnerabilities discussed above. Such an approach can be made scalable to hundreds of applications by simply combining the code-level analysis techniques that analyze any given application together with a system-level analysis techniques that analyze the library dependences of any given piece software in an OS. We discuss these techniques in detail in the next two sections.

3 Methodology

3.1 Problem Formulation

As mentioned earlier, our approach aims to find vulnerabilities regarding a client's incorrect use of APIs for hostname and certificate validation.

3.2 High-level Approach

Our overall approach is summarized in Figure 2. The client software is input to a static code analyzer which transforms the software to an abstract representation. The correct uses of the SSL/TLS library APIs are specified as signatures, and provided to the signature matching tool, which matches the signatures against the abstract representation of the software. If a match is found, the client software validates the hostname and the certificate correctly, and otherwise, a vulnerability is reported.

3.3 Code Representation

For representing the program, the static analyzer produces abstract representations. Many different graph-based code representations have been developed for code analysis. Our choice of code representation is driven by their support for reasoning about the types of vulnerability patterns that exist in the original code itself. Among code representations, the most common ones are control flow graph and data flow graph. We discuss their usefulness as program representations below.

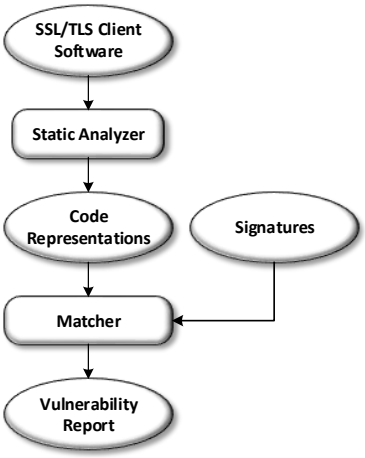

Fig. 2. Methodology

A *Control Flow Graph (CFG)* is a directed graph that captures the control-flow structure of a program, representing all the possible execution paths. Each node of a CFG represents a basic block which is a portion of the code with only one entry point and only one exit point. CFG also reflects the execution order for each node and the conditions to be satisfied to execute a particular path. CFGs are good in capturing temporal relationships between calls to functions or statements. For instance, in typical SSL/TLS application programmed using GnuTLS, the first certificate in the chain is chosen by the `gnutls_x509_crt_import` method, but this must be proceeded by the method `gnutls_certificate_get_peers` that gets the entire certificate chain. Such temporal relationships are captured by CFGs. However, reasoning about data flows in an application purely with CFGs is difficult.

To address the difficulty of reasoning about data flows in the application, a *Data Flow Graph (DFG)* may be used. A DFG is a directed graph which shows the data dependences between various objects, and the relationship between input to functions and their output values.

Let us consider a simple example that was introduced earlier. In order to reason about the output of `guntls_certificate_verify_peer2` for certificate validation, the return values of the function needs to go through a number of checks. Data flow graphs support reasoning about such 'reaching definitions', by preserving the *def-use* chains in the program.

The above discussion makes it clear that we need to reason about both control flow and data flow relationships in programs. Therefore, neither CFGs nor DFGs by themselves are sufficient. However, to reason about the two together, program representations such as *Program Dependence Graph (PDG) [10]* have been studied earlier and have been successfully used in analysis tools. Derived from the program's CFG and DFG, PDG summarizes both data dependences and control dependences among all the statements and predicates in the program.

The nodes of a PDG represent different statements or predicates of the procedure. As for the edges, generally PDG has two types of edges: control dependence edges and data dependence edges, which represent the control and data dependencies among the procedure's statements and predicates. For nodes X and Y in a PDG, Y is control dependent on X if, during execution, X can directly affect whether Y is executed. Also, X is data dependent on Y if Y is an assignment and the value assigned in Y can be referenced from X. Each PDG represent the code structure within a procedure and different PDGs can be interconnected together to reflect the code structure of the whole program.

In summary, compared with a control flow graph, PDG explicitly represents the essential control relationships implicitly presented in the control flow graph. In addition, it also explicitly represents data flow relationships of the program. This simplifies the task of reasoning about vulnerability patterns that involve both control and data flows.

3.4 Vulnerability Identification

The problem of vulnerability identification mentioned above can compactly be summarized as follows: given a PDG of a client application that is using SSL library APIs, how to automatically locate any vulnerabilities in the use of SSL APIs with good efficacy and accuracy. Before presenting our matching approach, we first review some examples of how SSL library APIs typically are invoked for certificate validation, and the kinds of patterns they constitute.

3.5 Example Patterns in the use of SSL APIs

For software using OpenSSL, certificate validation is done by a series of API function calls, each of which may closely related to others in terms of data flows and control flows. The correct use of such APIs can be abstracted as API patterns. In an SSL application, a failure to follow such patterns can consequently lead to a vulnerability.

Generally, a basic validation of SSL/TLS certificate should include the following steps: (1) verify that the certificate is signed by the trusted CA; (2) verify that the signature is correct; (3) verify that the certificate is not expired; and (4) verify that the `CommonName` of X.509 certificate and the domain name (hostname) matches.[1] As a result, certain patterns should be followed when programming with OpenSSL APIs.

By default, OpenSSL performs a built-in certificate validation during SSL/TLS handshake but ignores any encountered errors. The application is therefore required to check the result of the validation after the handshake and drop the connections if necessary before communicating over SSL/TLS (as shown in Listing 2). The API function `SSL_get_verify_result` (at line 20 in Listing 2) returns a macro value `X509_V_OK` when the validation succeeds. According to OpenSSL

1. (1)(2)(3) are referred to P2 and (4) is referred to P1 in Section 2.1.

document [11], one design flaw of this API function – often neglected by developers – is that the function also returns `X509_V_OK` when there is no peer certificate presented and thus no validation errors occurring in such case. As a consequence, `SSL_get_verify_result` should be used only together with another API function: `SSL_get_peer_certificate`, to check whether a peer certificate is presented.

Besides this, OpenSSL also provides an API function `SSL_CTX_set_verify` to configure this built-in certificate validation, which is typically performed during the handshake (See Figure 1). The handshake is immediately terminated if the built-in certificate validation fails, and if the `SSL_VERITY_PEER` flag is set to this function (as shown in Listing 1). In this way, further checks of validation result will not be necessary any more. In addition, `SSL_CTX_set_verify` also provides a callback function to modify the built-in validation results for every single certificate in certificate chain. This callback function allows applications to add customizations to the built-in validation process.

3.6 Design Space for Signatures

Vulnerability Signatures vs. Correct-use Signatures SSLINT is to detect incorrect use of SSL APIs in an application by looking for patterns (that we call signatures) in its code. In order to do this, we have the choice of proceeding in two ways. The first is to model *incorrect* uses of the API by an application and look for matches in the application. This way, the returned matches will constitute possible vulnerabilities. The main drawback of this approach is the difficulty of getting a *complete* description of the ways in which a vulnerability could manifest. In order to achieve that, the signature developer needs to anticipate all possible ways in which the programmer of the SSL application could incorrectly use the API, clearly an uphill task. Furthermore, failure to model any incorrect uses may result in *missed vulnerabilities* by our approach.

The second approach, the one adopted in this paper, is to model *correct-uses* of the SSL APIs for hostname and certificate validation, and look for whether these signatures are matched in the application code. In this approach, the signature developer comes up with the patterns of how to correctly use the API in order to perform hostname and certificate validation. Then an automated approach can look for whether the application matches these correct usage patterns, and report any mismatches. The advantage of this approach is that the typical number of ways of correctly using these APIs is small, and therefore it is possible to come up with a precise signature to characterize the correct use of the API. Furthermore, an incomplete specification *does not* result in missed vulnerabilities, but only manifest as false alarms. By carefully examining the false alarms from some initial deployment of the tool, we can eliminate them and make the tool to be precise, a fact that we will discuss in the evaluation.

For example, in Listing 2, we need to model API patterns and convey the logic behind these patterns in our signature. Specifically, first, the return values of `SSL_get_peer_certificate` at line 18 determines which branch should be taken in the program, so does the `SSL_get_verify_result` at line 20. Second, `ssl` is defined by `SSL_new` at line 13 and used by `SSL_connect` at line 16, `SSL_get_peer_certificate` at line 18 and `SSL_get_verify_result` at line 20. It is similar for `ctx` at line 10 and `SSL_new` at line 13.

Signature Representation To model these aforementioned patterns, many types of signature representations can be used and some common ones include regular expressions [12], [13], state machines [14]. Brumley et al. made the important observation that signatures could be represented across a spectrum of complexity classes [15].

To represent correct-use signatures, one can think of using regular expressions. We first note that regular expressions are good for matching temporal sequences of function calls. Unfortunately, they do not work well for patterns that involve data flows.

For example, consider the def-use chain (Shown in Listing 2). Matching parameters or variables alone is insufficient for verifying the correct use of these API calls, we need to link the output of `SSL_get_verify_result` for certificate validation, with subsequent checks that use this return value, factoring for data flows.

Another signature data structure involves the use of protocol state machines. Some of these state machines are strictly more powerful than regular expressions. Some of these signature representations are used to match inputs (e.g. network traffic), and have the expressiveness of Turing machines. For a static analysis approach such as SSLINT, they are inherently unsuitable, as the corresponding decision problem that involves matching such a Turing signature and a program is undecidable.

Our representation Our choice for signatures are labeled graphs, a simpler representation for our signatures. Our signature graph involves nodes that represent instructions in the code and edges that represent correlations between different nodes. The signature reflects the correct use of the API to be matched in the code, including critical API call-sites, variables, parameters and conditions. Using recent advances in graph mining, we also use graph query language [16], a concept widely used in graph databases, to describe our signature and explain how the signatures are matched in real code.

3.7 Matching Procedure

Given that we have a program representation in the form of a PDG, and a signature represented in the form of a labeled graph, the matching procedure can be done in several ways.

A first choice is to treat the PDG as a labeled graph, and specify the signature at a higher level of abstraction

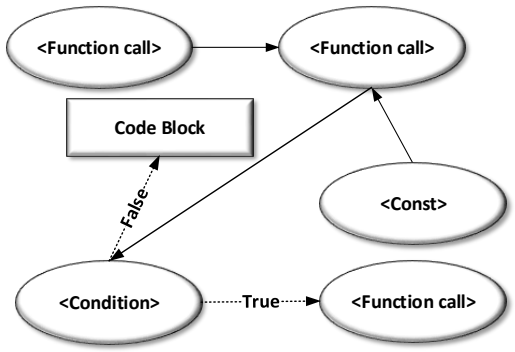

Fig. 3. Signature based on PDG.

(e.g. the return value X of a method f, flowing to a call site g). In this case, we need to develop a matching algorithm for searching this high level signature pattern in the labeled graph. The second approach is to treat the PDG as a simple labeled directed graph, and specify the signature in terms of the nodes and edges of this labeled graph and invoke a graph matching procedure that looks for this signature in the PDG of the program. The advantage of the latter approach is that we can make direct use of graph query languages to encode signatures and make use of matching procedures designed efficiently for them. In the rest of this section, we describe this approach.

For the sake of illustration, we also present our signatures as a PDG. Figure 3 shows a simple PDG-based signature, in which solid arrows represent data dependences while dotted arrows represent control dependences. One important distinction between a program's PDG and the one use to represent its signature (as in Figure 3) is that data dependences between two nodes (noted in solid arrows) in signature do not necessarily mean that they are adjacent neighbors in the program's PDG. It only reflects the fact that they are start and end points of a data flow and there are possible intermediate nodes along the data flow in the PDG of code.

To illustrate our signature matching approach, we use a graph query language to specify the matching approach in a declarative manner. In particular, we discuss how the PDG based signatures are represented in Cypher. (Cypher is a declarative, SQL-inspired language for describing patterns in graphs supported by the popular graph database Neo4j.) Cypher allows users to describe what they want to select, insert, update or delete from a graph database. For simplicity, we describe our signatures using a simplified Cypher style graph query language in Equation (1). The key abstraction in this language is the $MATCH$ predicate, which specifies the nodes, edges as well as labels on edges to be matched in the query. For example, $(v_1) \rightarrow [data](*) \rightarrow (v_2)$ represents a data dependence from node v_1 to v_2 in a PDG. The optional asterisk after the edge label matches both direct and indirect dependences. The $WHERE$ predicate specifies all the conditions of the match, including properties of nodes and edges. The $RETURN$ predicate acts as a filter and specifies what should be returned from the matching result.

A Cypher style query is thus generally written as:

$$\begin{array}{ll} MATCH & (v_i) \rightarrow [l](*) \rightarrow (v_j) \\ WHERE & [condition] \\ RETURN & v_i, v_j \end{array} \quad (1)$$

Note that the final result of such a query is a set of all tuples that satisfy the conditions in the $MATCH$ and $WHERE$ clauses. By describing a PDG-based signature in Cypher style, our signature matching algorithm can be interpreted to performing queries on PDG of a target program, and triggering an alert whenever there queries do not return any result. In next subsection, we present an intuitive example to show how we develop signature for OpenSSL client applications and how the matching algorithm works with the signature.

3.8 Signature Development

As shown in Listing 1 and 2, multiple APIs are involved in the certificate validations. Any incorrect use of these critical APIs could make an application vulnerable to MITM attacks. To model these API patterns as the first step of automatic vulnerability detection, we design a signature so that all the API patterns are correctly extracted in the form of control and data dependences.

In OpenSSL, data structures such as `SSL_CTX` and `SSL` are involved in most APIs for certificate validations. So data flow dependences between these APIs, need to be modeled in the signature so that data flows belonging to different sessions (such as for servers and clients) are extracted correctly. For APIs `SSL_get_verify_result` (Line 20 in Listing 2) and `SSL_get_peer_certificate` (Line 18), the signature needs to model both the data flow dependences such as return values and the control flow dependences such as different execute paths.

In addition, the signature also needs to model the control dependences between certificate validation APIs and SSL read/write APIs. It is because an SSL/TLS client should not read or write any application data until the client confirms the server's identify by certificate/hostname validation; otherwise the client is vulnerable to MITM attacks (See Section 2.3). In particular, if the certificate/hostname validation happens after the SSL/TLS handshake (e.g., in Listing 2), such vulnerable API uses are possible.

Algorithm 1 Signature Matching Algorithm.

```
1: R := executeQuery(Query_0)
2: for (m, n) ∈ R do
3:     if ⋃_{i>0} executeQuery(Query_i(m, n)) = ∅ then
4:         alert("Vulnerability Detected.")
5:     end if
6: end for
```

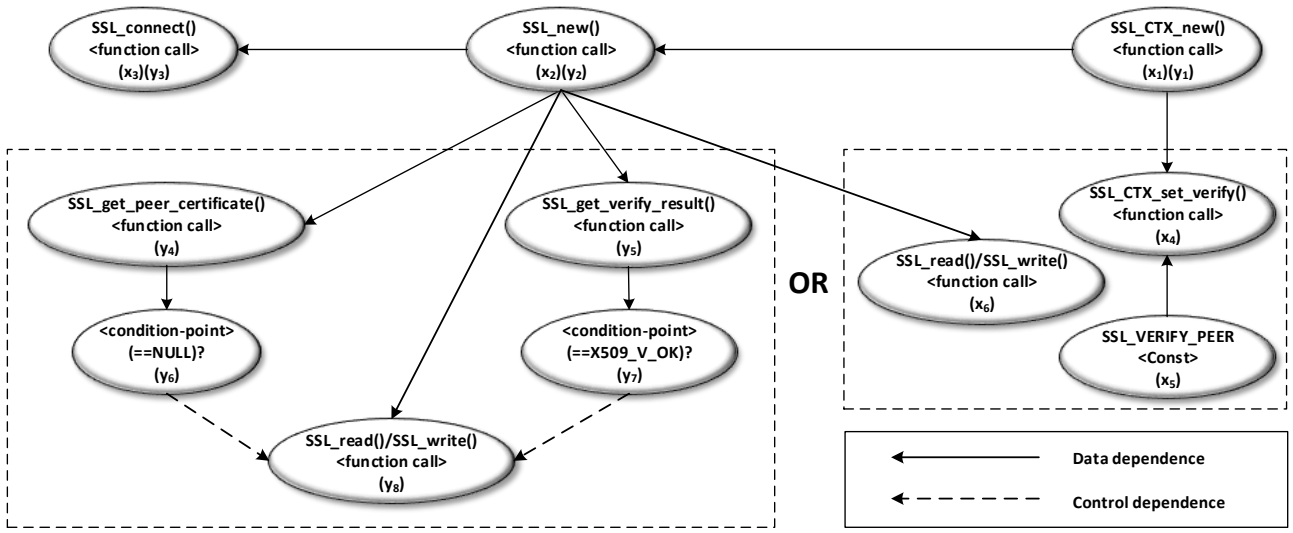

Fig. 4. Control and data dependences representing Listing 1 and Listing 2. These dependences must be captured in our signature queries.

Figure 4 specifies these above-mentioned dependences for OpenSSL validation API in Listing 1 and Listing 2. Obviously, there is some overlap between the two patterns (different part is marked with dashed boxes), so actually there are two sub-signatures in Figure 4 and either of them represent a correct logic for certificate validation in SSL/TLS client application.

Given the dependences, it is now easy to develop our signature queries and the signature-matching algorithm. First, we need to find all the candidate sessions whose validation must be checked. The data dependences from the initialization API calls (such as `SSL_new()`) to the send/receive API calls (such as `SSL_write()` and `SSL_read()`) represent exactly these sessions. We can collect all such dependences with the following Query_0.

Query_0:

$$
\begin{aligned}
&MATCH \\
&\quad (m) \rightarrow [data]* \rightarrow (n); \\
&WHERE \\
&\quad m.callsite == SSL\_new() \quad AND \\
&\quad (n.callsite == SSL\_read() \quad OR \\
&\quad n.callsite == SSL\_write()) \\
&RETURN \\
&\quad m, n
\end{aligned}
\tag{2}
$$

Given the result of Query_0, we can now match all the dependences depicted in Figure 4 with the following two parameterized queries.

$\text{Query}_1(M, N)$:

$$
\begin{aligned}
&MATCH \\
&\quad (x_1) \rightarrow [data]* \rightarrow (x_2); \\
&\quad (x_1) \rightarrow [data]* \rightarrow (x_4); \\
&\quad (x_2) \rightarrow [data]* \rightarrow (x_3); \\
&\quad (x_2) \rightarrow [data]* \rightarrow (x_6); \\
&\quad (x_5) \rightarrow [data]* \rightarrow (x_4); \\
&WHERE \\
&\quad x_1.callsite == SSL\_CTX\_new() \quad AND \\
&\quad x_2 == M \quad AND \\
&\quad x_3.callsite == SSL\_connect() \quad AND \\
&\quad x_4.callsite == SSL\_CTX\_set\_verify() \quad AND \\
&\quad x_5.type == const \quad AND \\
&\quad x_5.value == \text{“}SSL\_VERIFY\_PEER\text{”} \quad AND \\
&\quad x_6 == N \\
&RETURN \\
&\quad x_1, x_2, x_3, x_4, x_5, x_6
\end{aligned}
\tag{3}
$$

$\text{Query}_2(M, N)$:

$$
\begin{aligned}
&MATCH \\
&\quad (y_1) \rightarrow [data]* \rightarrow (y_2); \\
&\quad (y_2) \rightarrow [data]* \rightarrow (y_3); \\
&\quad (y_2) \rightarrow [data]* \rightarrow (y_4); \\
&\quad (y_2) \rightarrow [data]* \rightarrow (y_5); \\
&\quad (y_4) \rightarrow [data]* \rightarrow (y_6); \\
&\quad (y_5) \rightarrow [data]* \rightarrow (y_7); \\
&\quad (y_6) \rightarrow [control] \rightarrow (y_8); \\
&\quad (y_7) \rightarrow [control] \rightarrow (y_8); \\
&WHERE \\
&\quad y_1.callsite == SSL\_CTX\_new() \quad AND \\
&\quad y_2 == M \quad AND \\
&\quad y_3.callsite == SSL\_connect() \quad AND \\
&\quad y_4.callsite == SSL\_get\_peer\_certificate() \quad AND \\
&\quad y_5.callsite == SSL\_get\_verify\_result() \quad AND \\
&\quad y_6.condition == \text{“} == NULL\text{”} \quad AND \\
&\quad y_7.condition == \text{“} == X509\_V\_OK\text{”} \quad AND \\
&\quad y_8 == N \\
&RETURN \\
&\quad y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8
\end{aligned}
\tag{4}
$$

Note the presence of parameters M and N in Query_1 and Query_2. These are the results of Query_0, plugged into Query_1 and Query_2, so that we can ensure we are matching API calls related to a particular session only. We also point out that we need two queries, Query_1 and Query_2, for matching because there are two correct validation logic patterns for OpenSSL. In case of GnuTLS, there is only one logic and so we will have only one query.

The general signature matching algorithm is thus as specified in Algorithm 1. Recall that the result of a query matching is a set of tuples. The for loop in line 2 iterates over all (m, n) tuples and executes queries Query_1 through Query_k (for OpenSSL $k = 2$), substituting parameters M and N by m and n respectively. If none of the queries return a non-empty set, the match failed, implying the absence of correct logic and presence of a vulnerability.

3.9 Code Pruning

Recall that our signatures are based on program dependence graphs, which must be generated using sophisticated static analysis. Static analysis in general is time consuming and may not scale well to large code. In order to mitigate the scalability issue, we designed a preprocessing step that takes as input a roughly (inaccurately but quickly) computed call graph of the application and outputs a smaller part of the application that is relevant for SSL API vulnerability analysis.

From the call-graph we locate nodes that are related to SSL APIs and trace calls to them all the way to public APIs or the main function. Any functions encountered on these paths are included for subsequent analysis; all other functions are excluded. Due to our limitation in specifying to our underlying static analysis tool (CodeSurfer) the exact functions to analyze, we instead include the files (source or headers) declaring and defining those functions.

3.10 Analysis of Libraries

Recall that our signatures are based on program dependence graphs, which must be generated using sophisticated static analysis. Static analysis in general is time consuming and may not scale well to large code. In order to mitigate the scalability issue, we designed a preprocessing step that takes as input a roughly (inaccurately but quickly) computed call graph of the application and outputs a smaller part of the application that is relevant for SSL API vulnerability analysis.

From the call-graph we locate nodes that are related to SSL APIs and trace calls to them all the way to public APIs (library programs) or the main function (client programs). Any functions encountered on these paths are included for subsequent analysis; all other functions are excluded. Due to our limitation in specifying to our underlying static analysis tool (CodeSurfer) the exact functions to analyze, we instead include the files (source or headers) declaring and defining those functions.

3.11 Analysis of Libraries

Libraries are partial programs or modules intended to be used in other programs. Vulnerabilities in a library ultimately reflect as vulnerabilities in the client applications. The analysis of libraries, i.e., partial programs or modules intended to be used in other programs, poses two significant challenges:

First, even though a library may have incorrect usage of SSL APIs, the data flows across the SSL APIs may happen only through the library API, implying that a client program using the library is necessary to complete the data flow. Since there is no obvious way of developing client programs that use the libraries in intended ways, we resolve this challenge in the following ways: (a) Open source libraries often have test cases or sample code distributed together with source code. Making use of these existing program can avoid developing new ones, which makes our analysis complete. (b) For those do not provide any test cases or sample code, we leverage on existing package management repositories of operating systems (Section 4.1) to resolve all the client programs which have dependences on these libraries and put the client program and library together to make our analysis complete. (c) For those (a) and (b) do not apply, we manually developing client programs that complete these data flows. The libraries and the client programs are then analyzed together for detecting vulnerabilities. Considering an example library `LibX`, which uses OpenSSL as its underlying library and has 2 library APIs: `LibX_initialize_session`, and `LibX_start_connection`. The OpenSSL API `SSL_CTX_new` and `SSL_new` are in library API `LibX_initialize_session`, while `SSL_connect` is in `LibX_start_connection`. Here we need to develop a test client program calling both `LibX_initialize_session` and `LibX_start_connection` to complete dataflow for analysis.

The second challenge is that even if a vulnerability exists in a library and can be detected by just the static analysis of the library alone, it does not imply if the vulnerable code will be reachable from the public library APIs. We therefore need to conduct a reachability analysis for the vulnerable section of the code. We do this by starting from PDG nodes indicating SSL API usage and tracing back from them by control dependencies. If we can find the library's public APIs on these paths, the vulnerable code section is reachable.

4 Implementation

This section describes the implementation of SSLint as a robust and scalable automated framework for vulnerability detection in C/C++ source code as well as other artifacts needed for the measurements covered in the next section. Our implementation of SSLint takes about 2600 lines of C/C++ code. In this section, we first introduce

the techniques for selecting candidates for vulnerability analysis, then we describe the implementation details of the static analysis on which our signature matching is based. Finally, we detail the techniques we used to verify the result of automated signature matching through manual auditing.

4.1 Candidate Selection

The first question to answer before the implementation is how to find the software using specific SSL libraries. The vulnerability matching only makes sense in software using SSL libraries. We leverage the data from package management repositories maintained by many Linux distributions and other communities. Many Linux distributions such as Ubuntu, Fedora, and OpenSuse have their own freely accessible software repositories, maintaining a large majority of common software, including SSL libraries, for distribution within their own ecosystems. Third-party software repositories also exist for Mac OS. All package management repositories commonly provide version control and information about package dependences for each software package. We leveraged information about package dependences to search for all software that depend on specific SSL libraries.

For our measurements, we used Ubuntu's official software repositories. To consider an example, the OpenSSL library is listed there as `libssl`[2]. After this small manual annotation, we were able to search dependence attributes for all packages and automatically list candidates that depend on OpenSSL.

It is noteworthy that the above approach can only detect packages that use SSL libraries via dynamic linking. However, this is not a fundamental limitation of our approach: to do a complete search, covering usages via static linking as well, we could instead search for specific SSL library headers in the package source code.

4.2 Static Analysis

This section briefly describes the core components of static analysis and other details needed for a working SSLINT.

4.2.1 Core components

We leverage CodeSurfer [17] for our static analysis. It is a tool for understanding of C/C++ programs. It supports deep semantic static analysis of programs and queries for understanding the source code. Apart from being a code-understanding tool, CodeSurfer is also a platform on which to build other advanced analyses. CodeSurfer generates and exposes to the users a series of program representations, including Abstract Syntax Trees (AST), Control Flow Graphs (CFG) and Program Dependence Graph (PDG), as a basis for further analysis.

2. There are both `libssl0.9.8` and `libssl1.0.0` packages in Ubuntu, and here we use `libssl` for simplicity.

Our static analysis begins by parsing the program and preparing an intermediate representation out of it. Then a control flow graph (CFG) on this intermediate representation and a class hierarchy analysis is performed. Following these analyses, we do a pointer analysis, which maps all pointers to possible abstract memory locations. Pointer analysis and call-graph construction work together and at the end of the analysis, function pointers and virtual function call targets can be resolved. We specifically use Andersen's pointer analysis [18]. Our analysis is field-sensitive (it can distinguish between different fields of the same object), flow-insensitive (instructions within a function treated as an unordered collection), and context-insensitive (it can not differentiate among calling contexts of a procedure). Finally, based on the call graph and pointer information, an interprocedural data flow analysis can be performed. This analysis together with the control flow information is then used to construct the PDGs.

Recall that prior to the actual static analysis, we perform a code pruning step. This step also utilized CodeSurfer, specifically running it in "super-lite" mode. Unfortunately, CodeSurfer, a proprietary tool, does not provide details of the analysis options used in "super-lite" to compute the application call graph. Nonetheless, we found the computed call graph to be sufficient for our purposes.

As a platform for static analysis, CodeSurfer provides APIs that expose its program representations. We implemented our client analyses as a plugin using these APIs to access call graphs and PDGs generated from a program. With that said, our approach of PDG-based signature matching for vulnerability detection is general and may be used for any programming language. For example, our technique could be made to target Java using static analysis frameworks such as WALA [19].

4.2.2 Automated building

A successful static analysis depends on the ability of the tool to understand code organization, e.g., which headers get included in which files, and where the definitions of functions declared in the headers can be found. This information is already available in build scripts, such as makefiles.

CodeSurfer emulates the interfaces of several standard C/C++ compilers (such as `gcc`) to serve as a drop-in replacement for the standard compilers in the build scripts. In this way, it is able to leverage the existing build system to understand code organization.

To provide an automatic build system for every software package we analyze is challenging: different pieces of software use different build systems such as cmake, autotools, make, and so on. With no common standard, it is difficult to build packages automatically. The situation is further complicated when the build needs specific libraries with possibly specific versions installed on the system. Finally, packages may need special configuration, including setting of compilation flags.

TABLE 1
Library model defined for OpenSSL and GnuTLS APIs.

OpenSSL	GnuTLS
SSL_CTX_new()	gnutls_init()
SSL_new()	gnutls_credentials_set()
SSL_get_peer_certificate()	gnutls_certificate_get_peers()
SSL_get_verify_result()	guntls_certificate_verify_peer2()
SSL_CTX_set_verify()	gnutls_x509_crt_import()
SSL_connect()	gnutls_x509_crt_check_hostname()
	gnutls_handshake()

Listing 4. Library model of SSL_new.

```
1 SSL *SSL_new(SSL_CTX *ctx)
2 {
3     SSL *s;
4     //standard memory allocation
5     s=(SSL *)malloc(sizeof(SSL));
6     s->ctx=ctx;
7     return s;
8 }
```

To meet this challenge, we again take advantage of package management tools and repositories. Tools such as yum (for Red Hat-based Linux distributions) and apt (for Debian-based distributions) not only allow installation of packages from online repositories but can also be used to download package source code, compile it, and then install the binaries. The repository maintainers have already integrated the build processes into a common interface understood by package management tools. We leverage this common interface to completely automate the build processes. For the work presented in this paper, we used the Ubuntu package managers. The following Ubuntu commands can be used to resolve all the building dependences and configuration for any package in the software repository.

```
apt-get -y build-dep {Package Name}
apt-get source {Package Name} --compile
```

4.2.3 Library Modeling

Software is rarely self-contained. Most software have external dependences such as libraries. In static analysis, the whole picture cannot usually be painted with the code of target software alone. With the absence of the code from other relevant component, tracking interprocedural data dependences is often impossible because the analyzer has no idea what a certain library function does inside its body.

A naïve approach to find these missing dependences is to integrate all the relevant code for analysis. However, this approach would greatly increase the amount of code to analyze and thus reduce scalability of the analysis. Therefore, a routine technique is to simply provide models for the external code, which adequately summarize the effects of the external code for the purpose of the analysis. For our case, we model the dependence properties of functions in libraries.

CodeSurfer [17] provides basic library models for API functions in standard system libraries (e.g. printf()), but it is far from complete. But it is also difficult to create accurate library models for a general used software (i.e. software for Unix-like OS) by analyzing the code in all relevant libraries. Thus certain kind of approximation need to be made. In CodeSurfer, the default model for undefined functions is that the return value data depends on the values of all actual parameters, but dependence on non-local values and return of pointer values are both ignored. Such approximation will possibly bring false positive and false negative. While we retain the default model, we add custom models for SSL/TLS library functions related to certificate validation and hostname validation (Table 1).

Listing 4 shows how we model the library function SSL_new. Compared with the original code, this model only keeps the data dependence between the parameter ctx and the return value. Besides, it also returns a heap variable allocated by a standard memory allocator. This fact is important for pointer analysis, which is used to generate data dependence edges in PDG. By applying library models, the analyzer gets a complete view of the code at hand while not worrying about the complexities in external code.

4.3 Signature Matching

Based on PDG structures output from CodeSurfer, we develop an implementation of the signature matching algorithm as described in Sections 3.7 and 3.8. Rather than using a graph database system like Neo4j, we use a custom implementation of traversal and querying of the program PDG that realizes Algorithm 1.

4.4 Manual Auditing

To verify the vulnerabilities reported by SSLINT, we take a dynamic approach to see if a software is really vulnerable to MITM attack. Since SSL is widely used to protect different application level protocols (HTTP, FTP, POP3, SMTP etc.), we cannot set up a general attack server for all clients we tested. Instead, this task requires human effort in understanding how the software are typically run. For this, we referred to the documentation accompanying the software and other online resources. Once it is clear how to run the software, the MITM attack situation itself may be emulated automatically. Rather than performing a real attack with, for example, an MITM proxy, we had the following simplified emulation of the attack.

Testing certificate validation: A standard certificate validation checks whether the certificate is expired. As a result, we can simply change the system time to sometime in the future to guarantee all the certificates to be expired, for example, the year 2099. If a successfully establishment of an SSL connection initiated by a client is observed, then we consider the client vulnerable to MITM attacks.

Testing hostname validation: We change the local DNS record by modifying `hosts` file and redirect the client we tested from a legitimate server to another. For example, we can redirect a SMTP client which intended to visit smtp.gmail.com to another SMTP server. A successful connection implies a vulnerability.

We also use Wireshark [20] as a sniffing tool between client and server to make sure if an SSL connection is established with no error. In summary, our manual auditing is done on a client machine, and no proxies are needed because we just want to prove the possibility of MITM attacks rather than actually perform the attack, which simplifies the auditing process.

5 RESULTS

This section describes our results from a large-scale automated signature-based SSL/TLS vulnerability detection on Ubuntu 12.04 open-source software packages using SSLINT.

5.1 Experimental Setup and Results Summary

We applied SSLINT to find vulnerabilities in software using OpenSSL or GnuTLS, which are the two most popular SSL/TLS libraries. In all, we found 593 software packages using these libraries (455 depend on OpenSSL only, 136 depend on GnuTLS only and 2 depend on both according to Ubuntu) out of 40636 in Ubuntu source list using candidate selection techniques described in Section 4.1. Among the 593 packages, 485 are SSL/TLS clients and 108 are libraries leveraging on OpenSSL or GnuTLS. We used a Linux server with a 2.2 GHz Intel Xeon CPU and 16GB memory for all our experiments. The analysis of these 593 packages amounts to analyzing over 30 million lines of C/C++ source code. Furthermore, we applied our code pruning technique in PDG construction. Overall, we successfully built PDGs from 492 packages. Other 101 failed due to static analyzer syntax error or memory explosion, which we will discuss in Section 6. The signature matching time for analysis of any package of the 492 is bounded by 120 seconds, showing a high efficiency of our approach.

Overall, we identified 29 previously unknown vulnerabilities (Shown in Table 2), which fall into 2 categories: certificate validation and hostname validation (Section 2.1). We further successfully performed MITM attacks on 23 of them through manual auditing (Section 4.4). Among the types of identified vulnerable packages are mail server, mail client, IRC client, web browser, database client, etc. Furthermore, we identified 9 false positives, which are caused by failures in data flow tracking in PDG. According to [11], API for hostname verification is currently unavailable in OpenSSL and will be supported in the future version 1.1.0. As a result, we only checked hostname validation for GnuTLS clients.

We reported all the vulnerabilities to Launchpad [21], the official bug tracker for Ubuntu software packages. Since most of vulnerable software we found in Ubuntu are community maintained and they are also distributed in other Linux distributions, the impact of these vulnerabilities we uncovered is beyond the scope of Ubuntu. For all the community-maintained software, we also reported the vulnerabilities to their upstream developers. So far, we have received 14 confirmations as well as a lot of interesting feedback, which will be discussed in the following subsections. The details of each vulnerability and the data compromise are illustrated in Table 2 and Table 3 respectively. We will next look at specific vulnerability cases.

5.2 SSL/TLS Vulnerabilities in Various Software

5.2.1 Mail Software

Email is one of the most important Internet applications. Emails themselves constitute highly private information for the users, so the security of email infrastructure is important. Unfortunately, our evaluation uncovered many unknown SSL/TLS vulnerabilities in mail software, which can lead to leakage of sensitive data such as email and user credentials or compromise of mail traffic integrity.

The email system is composed of mail clients and mail servers. An email is sent by a mail client or, more precisely, a Mail User Agent (MUA) to a sender's mail server, called Mail Transfer Agent (MTA), using SMTP protocol. Then the email is delivered to recipient's MTA by sender's MTA, again using SMTP. On receiving an email from another MTA, the recipient's MTA delivers the email to a mail box server, called Mail Delivery Agent (MDA), which stores emails for user and waits to receive. The recipient MUA can retrieve the email on a MDA using POP3 or IMAP protocols. Generally, a MDA requires a username and password for authentication when communicating with a MUA.

POP3S, IMAPS, and SMTPS are SSL/TLS-protected versions of the above protocols. According RFCs defining these protocol variants [22], [23], the mail client should check the server's identity by certificate validation as well as hostname validation during the handshake in order to prevent MITM attacks. Unfortunately, the following software fails to enforce this requirement.

Xfce4-Mailwatch-Plugin [24]: Xfce4 Mailwatch Plugin is a multi-protocol, multi-mailbox mail watcher for the Xfce4 panel, which acts as a simple mail client and generates notifications as soon as it receives new email from mail servers. According to Ubuntu Popularity Contest [25], it has 165,442 installs in total as of November 2014. It supports both POP3S and IMAPS. It uses GnuTLS for SSL/TLS implementation but fails to call `gnutls_certificate_verify_peers2` to check server's certificates after the successful establishment of a new SSL/TLS connection. Moreover, it also fails to enforce hostname validation. As a result, Xfce4 Mailwatch Plugin accepts any SSL/TLS certificate and an MITM attack can lead to *leakage of user credentials and emails as well as integrity violations for email messages*.

TABLE 2
Zero-day SSL/TLS vulnerabilities discovered by SSLINT in Ubuntu 12.04 packages.

Package Name[1]	Lines of Code	Client or Library	Type[2]	Underlying SSL/TLS Library	Location	Vulnerability Status
dma	12,504	Client	C	OpenSSL	/crypto.c	Confirmed
exim4	94,874	Client	H	OpenSSL/GnuTLS[3]	/src/tls-openssl.c /src/tls-gnu.c	Fixed
xfce4-mailwatch-plugin	9,830	Client	C/H	GnuTLS	/libmailwatch-core/mailwatch-net-conn.c	Proved[4]
spamc	5,472	Client	C	OpenSSL	/spamc/libspamc.c	Confirmed
prayer	45,555	Client	C	OpenSSL	/lib/ssl.c	Confirmed
epic4	56,168	Client	C	OpenSSL	/source/ssl.c	Fixed
epic5	65,155	Client	C	OpenSSL	/source/ssl.c	Fixed
scrollz	78,390	Client	C/H	OpenSSL/GnuTLS[3]	/source/server.c	Fixed
xxxterm	23,126	Client	H	GnuTLS	/xxxterm.c	Confirmed
httping	1,400	Client	C	OpenSSL	/mssl.c	Confirmed
pavuk	51,781	Client	C	OpenSSL	/src/myssl_openssl.c	Confirmed
crtmpserver	57,377	Client	C	OpenSSL	/thelib/src/protocols/ssl/outboundsslprotocol.cpp	Confirmed
freetds-bin	80,203	Client	C/H	GnuTLS	/src/tds/net.c	Confirmed
picolisp	14,250	Client	C	OpenSSL	/src/ssl.c	Fixed
nagios-nrpe-plugin	3,145	Client	C	OpenSSL	/src/check_nrpe.c	Confirmed
nagircbot	3,307	Client	C	OpenSSL	/ssl.c	Proved[4]
citadel-client	56,866	Client	C	OpenSSL	/utillib/citadel_ipc.c	Proved[4]
mailfilter	4,773	Client	C	OpenSSL	/src/socket.cc	Proved[4]
suck	12,083	Client	C	OpenSSL	/both.c	Proved[4]
proxytunnel	2,043	Client	C/H	GnuTLS	/ptstream.c	Proved[4]
siege	8,581	Client	C	OpenSSL	/src/ssl.c	Proved[4]
httperf	6,692	Client	C	OpenSSL	/src/core.c	Proved[4]
syslog-ng	115,513	Client	C	OpenSSL	/tests/loggen/loggen.c	Proved[4]
medusa	18,811	Client	C	OpenSSL	/src/medusa-net.c	Proved[4]
hydra	23,839	Client	C	OpenSSL	/hydra-mod.c	Proved[4]
ratproxy	4,069	Client	C	OpenSSL	/ssl.c	Proved[4]
dsniff	24,625	Client	C	OpenSSL	/webmitm.c	Proved[4]
libcyrus/cyrus-clients	143,768	Library	C	OpenSSL	/lib/imclient.c	Proved[4]
libofetion1	8,251	Library	C	OpenSSL	/fetion_connection.c	Proved[4]

[1] For simplicity, some rows indicate several similar packages.
[2] "C" is an abbreviation of "certificate validation" and "H" is an abbreviation of "hostname validation" (See Section 2.1). We do not check hostname validation for OpenSSL clients because there is no supported API.
[3] These packages actually depend on both OpenSSL and GnuTLS in code, but according to package dependence information provided by Ubuntu source list, they only have dependences on GnuTLS.
[4] We successfully proved these software to be vulnerable to MITM attack by performing dynamic manual auditing (See Section 4.4).

Mailfilter [26]: Mailfilter is a mail client utility for filtering out spam mails. It connects to mail server using POP3 or POP3S protocol, compares mails inside the mailbox to a set of user defined filter rules and deletes spam directly on the mail server. As a mail client, Mailfilter stores user credentials and user defined filter rules in its configuration files and uses OpenSSL as SSL/TLS implementation. But it neither calls `SSL_get_verify_result` after SSL/TLS handshake nor sets `SSL_VERIFY_PEER` flag before the SSL handshake, for necessary certificate validation. Consequently, Mailfilter can also lead to *confidentiality and integrity violation of emails and user credentials.*

Exim [27]: Exim is a popular message transfer agent (MTA) for use on Unix-like systems connected to the Internet. Statistics from Ubuntu Popularity Contest [25] show that the exim4 package has 112,530 installs as of November 2014. As discussed earlier, the SMTP protocol is used in two situations: 1) between a MUA and a MTA, and 2) between MTAs. When using SSL/TLS to protect SMTP protocol, the MTA acts as an SSL/TLS server to a MUA and an SSL client to other MTAs. Exim implements SMTP over SSL/TLS using both OpenSSL and GnuTLS and provides multiple options for users. Unfortunately, both implementations fail to enforce hostname validation during SSL/TLS handshake. In practice, networking situation between different MTAs varies greatly and thus MTAs cannot rely on insecure DNS. Attackers can possibly perform MITM attack or just hijack the SSL/TLS connection to a malicious host, *leakage or alteration of emails for a mass of users* using the MTA. We reported this vulnerability to Exim developers, who fixed it in version 4_83_RC1 by adding the `tls_verify_cert_hostnames` option to enforce hostname validation. Meanwhile, the developers also pointed out that a better solution to secure DNS for MTAs is in the DANE SMTP specification [28], which is not yet standardized.

DragonFly Mail Agent [29]: Like Exim, DragonFly Mail Agent (DMA) is another MTA. It supports SMTPS and uses OpenSSL for the implementation. DMA fails to enforce certificate validation and thus accepts any certifi-

TABLE 3
Possibly Compromised Data in Vulnerable SSL/TLS Software.

Vulnerable Software	Possibly Compromised Data
dma	Email contents.
exim4	Email contents.
xfce4-mailwatch-plugin	Email account and password.
libcyrus/cyrus-clients	Email account, password and email contents.
spamc	Email contents.
prayer	Email account, password and email contents.
epic4	Personal information and chatting logs.
epic5	Personal information and chatting logs.
libofetion1	Personal information.
scrollz	Personal information and chatting logs.
xxxterm	Web contents.
httping	Web server statistic information.
pavuk	Web contents.
crtmpserver	Video stream contents.
freetds-bin	SQL server user account, password, database contents.
picolisp	Any data sent to or received from the picoLisp server.
nagios-nrpe-plugin	Monitoring information of servers.
nagircbot	Monitoring information of servers.
citadel-client	Personal information such as email, chatting logs, etc.
mailfilter	Email account, password and email contents.
suck	Newsfeed.
proxytunnel	Any data in the SSL/TLS tunnel.
siege	Performance information of websites.
httperf	Performance information of websites.
syslog-ng	System logs of servers.
medusa	Data in password dictionary[1]
hydra	Data in password dictionary[1]
ratproxy	Data for security auditing[2]
dsniff	Data for security auditing[2]

[1] Medusa and hydra are both network logon crackers.
[2] Ratproxy and dsniff are tools for security auditing or penetration testing.

cates from other MTAs, making itself *vulnerable to email data leakage and alteration* under an MITM attack. The maintainers confirmed this vulnerability as we reported to them and they are fixing it now. However, they also point out that certificate validation is not always possible since some MTAs use self-signed certificates. This issue is further discussed in Section 5.4.1.

5.2.2 IRC Software

This section describes the vulnerabilities found in IRC clients. IRC is a multi-user real-time chat system. Users on an IRC channel can have real-time conversation with each other. Many IRC software use SSL/TLS to protect the communication between an IRC server and an IRC client, which makes them candidates for our search for certificate or hostname validation vulnerabilities.

Enhanced Programmable ircII client (EPIC) [30]: EPIC is a text-based ircII-based IRC client for UNIX-like systems and supports SSL/TLS for client-server communication. EPIC versions 4 and 5 leverage OpenSSL for SSL/TLS implementation but they only read the server certificate using `SSL_get_peer_certificate` rather than verify the certificate using `SSL_CTX_set_verify`, `SSL_get_verify_result` or custom functions. As a result, EPIC4/5 is vulnerable to MITM attacks leading to *leakage or change of IRC account information and chat messages*. EPIC maintainers promptly confirmed and fixed this vulnerability.

Scrollz IRC Client [9]: ScrollZ is another ircII-based IRC client, which also provides SSL/TLS support. ScrollZ supports both OpenSSL and GnuTLS by enabling different compilation flags. In function `login_to_server`, SSL/TLS is used for protect a username/password authentication when logging to an IRC server. Both the OpenSSL and GnuTLS implementations fail to validate server certificate, again leading to *leakage or modification of IRC account information and chat messages* under a MITM attack. This vulnerability is also confirmed and fixed.

5.2.3 HTTP Software

HTTPS, or HTTP protected by SSL/TLS, is widely supported and deployed. As a result, most common browsers do not have these security issues anymore. However, for non-browser applications, such vulnerabilities are still easy to find [2]. One of the vulnerabilities we identified in HTTP software is shown below.

Prayer [31]: Prayer is a webmail interface for IMAP servers (MUA) on Unix-like systems, which is comprised of a front end daemon, called prayer, and a backend daemon, called prayer-session. The frontend, prayer, is a simple HTTP server as well as a HTTP proxy that provides static web pages and forwards user requests to the backend, prayer-session, which handles communication with IMAP servers. Prayer-session inherits its IMAP implementation from an external library and the SSL/TLS connections between prayer-session and IMAP server are secure. However, the communication between the prayer frontend and prayer-session backend is not. Prayer-session communicates with the user using HTML over HTTP/HTTPS connections through the prayer proxy, which does not enforce certificate validation (use OpenSSL for implementation), making it vulnerable to MITM attacks with possible *confidentiality and integrity compromise of user credentials and email messages*. Although prayer and prayer-session is typically deployed on a loopback interface of the same machine, or on a trusted LAN, making the impact relatively low, there is still risk of sensitive data leakage. So far this vulnerability has been confirmed and the maintainer is now taking actions.

5.2.4 Other Software

In addition to the vulnerabilities described above, we also identified vulnerabilities in other software using less-common application layer protocols protected by SSL/TLS. Generally, SSL/TLS is a transport layer protocols and it can be used to protect any data in application layer. As a result, SSL/TLS is widely used in many

different types of software. One of the vulnerabilities we identified is in a database client.

FreeTDS [32]: FreeTDS is a set of open source clients and libraries for Unix-like systems that provide access to Microsoft SQL Server and Sybase databases. TDS stands for Tabular Data Stream, a protocol primarily used between Microsoft SQL Server and its client. Like other protocols of this kind, TDS protocol depends on a network transport connection established prior to a TDS conversation. TDS also depends on SSL/TLS for network channel encryption and authentication. Generally, Microsoft SQL Server can be configured with a server certificate for clients to verify its identity. This certificate can either be self-signed or a valid one signed by a trusted CA. FreeTDS uses GnuTLS for SSL/TLS implementation, but fails to enforce any kind of certificate validation or hostname validation, nor does it provide any kind of options for developers to do the validations, making TDS connections between a database client and a server vulnerable to MITM attacks. This vulnerability can lead to *confidentiality and integrity compromise of user credentials and database contents*. So far, the vulnerability has been confirmed and the maintainer has agreed to add options for all the validations. Besides, they also point out the situation when self-signed certificate is used, which will be discussed in Section 5.4.1.

5.3 Scalability Evaluation

We describe here the scalability of our tool. In particular, we construct PDGs for each software package with our techniques two times, once without code pruning, and once with code pruning. The results are depicted in Figure 5. As described in Section 3.9, when performing code pruning, we first build call graph of the application and eliminate all functions which are irrelevant for SSL API vulnerability analysis. In total, we build PDGs for 61 application packages, and as can be seen in Figure 5, code pruning significantly reduces the time of PDG construction thus tremendously speeds up the analysis. For example, the analysis time of the application named 'epic5' decreases by 99.96% from 56.93 hours to 81.88 seconds, so we have to use **logarithmic scale** for Y-axis on the left. Overall, code pruning speeds up the analysis by more than 100% in 33 software packages we tested. For most packages our analysis completes, but in some cases the analysis could not complete. This is a general limitation of any static analysis, and we have tried to mitigate it to some extent using our code pruning analysis. As for accuracy, our code pruning approach achieves the same accuracy as our previous approach through manual auditing process (see Section 4.4), so we we do not sacrifice accuracy for speed.

5.4 Other Interesting Findings

Apart from all the vulnerabilities we identified, our measurements also gave the following interesting insights.

5.4.1 Use of Self-signed Certificate

Generally, in Public Key Infrastructure (PKI), trust between two parties is maintained by a trusted CA. A valid certificate signed by a trusted CA can be used as a proof of holder's identity, and can also be verified by others when communicating using SSL/TLS. In practice, sometimes self-signed certificate are used instead due to the cost or other reasons. A self-signed certificate is a certificate signed with its own private key. Everyone can issue self-signed certificate, so usually it should not be trusted. A client which accepts self-signed certificate is probably vulnerable to MITM attacks. As many developers commented on our vulnerability report, there is no clear solution for self-signed certificate in general cases. As a result, self-singed certificate is not recommended in SSL/TLS, especially on sensitive, public connections. However, particularly, if both clients and servers are managed by one party or they are able to build trust through other channels, then signing a certificate with one's own CA can be a solution for those who unwilling to pay for a signed certificate.

5.4.2 Community Maintained Software in Linux Distributions

Our evaluation also reveals the "security gap" between upstream projects and packages in Linux distributions. For example, we analyzed 492 software packages in Ubuntu 12.04, many of which are community maintained software and have their own upstream projects. Usually, these software also have packages in other Linux distributions. Some vulnerabilities still appear in distribution packages even they have been fixed for years in upstream projects. For instance, we found a certificate validation vulnerability in a Ubuntu package (in all versions including the latest Ubuntu 14.10) named `imapproxy` [33], which was already fixed in its upstream in Jan. 2014. On one hand, the Ubuntu maintainers are usually not responsible for the community-maintained software, and one needs to first contact upstream developers if she finds a bug or vulnerability and then submit a patch to Launchpad [21], the official Ubuntu bug tracker. We submitted all of the vulnerabilities in Table 2 to Launchpad first, but got the following response for most packages, "Since the package referred to in this bug is in universe or multiverse, it is community maintained. If you are able, I suggest coordinating with upstream and posting a debdiff for this issue. When a debdiff is available, members of the security team will review it and publish the package." On the other hand, many upstream developers feel no obligation to fix bugs or vulnerabilities in Linux distribution packages as is evident in the response of one upstream project maintainer, "That is indeed true as I said I will look into this and fix it for the next release. I don't follow bugs reported to various distributions, there are way too many. It would be much better if you reported them directly. I am aware the SSL implementation is

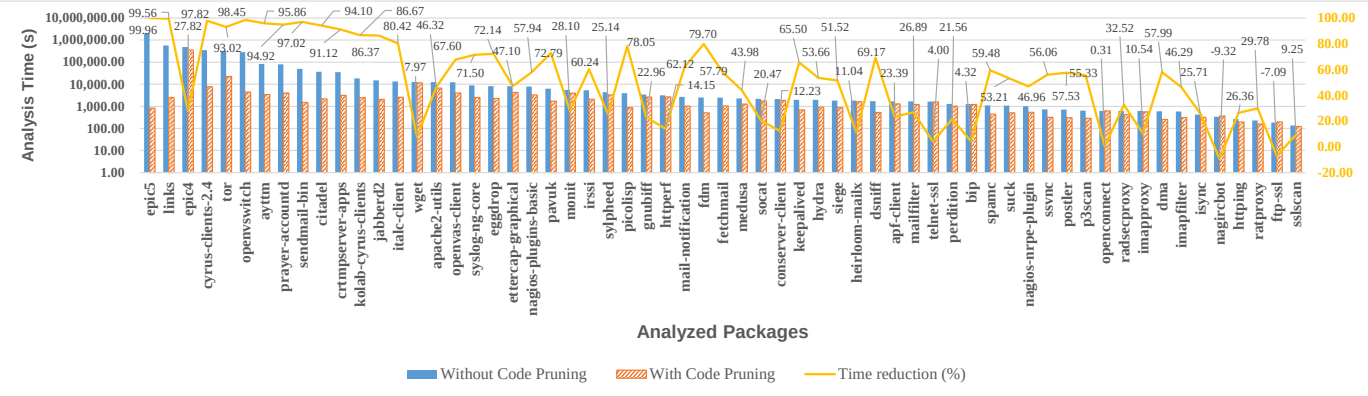

Fig. 5. Analysis Time Comparison.

bare bones. I will look into this and hopefully fix it for the next release." We think that explains why these distribution packages are of poor quality, and we believe that more efforts are needed to narrow down "security gap" by all community developers.

6 Limitations

Even though we showed SSLint's effectiveness at finding SSL usage vulnerabilities, we acknowledge the following limitations of our tool.

Static Analysis Accuracy. Static dependence analysis necessarily involves approximations, which may possibly lead to both false positives and false natives. In our implementation, we used CoderSurfer to construct PDG for our underlying analysis, which inevitably makes our results inherit the limitations from the implementation of CodeSurfer. In particular, we are aware that the following aspects of CodeSurfer would affect the precision and soundness of SSLint:

- *Aggregate variables.* Aggregate variables, such as arrays, unions and structures are modeled as a single variable, make SSLint prone to false positives.
- *Pointer analysis.* CodeSurfer adopts a flow insensitive, context-insensitive pointer analysis, leading to an over-approximation of PDG construction, again leading to a possibility of false positives in SSLint.
- *Reused memory.* Dependences between variables which share the same storage location are not modeled, leading to false negatives in SSLint.
- *Undefined functions.* Although CodeSurfer models popular C/C++ libraries such as libc and we also develop library models for some important API functions (Section 4.2), the modeling of libraries is far from complete. When a library function is undefined, indirect dependencies through pointer arguments, direct and indirect dependences through global and static variables are not modeled, leading to false negatives in SSLint.

Scalability. Apart from accuracy limitations, SSLint also inherits some scalability limitations from CodeSurfer. Inter-procedural analysis is computationally expensive. Based on our experience and observation, CodeSurfer usually has problems generating PDGs for software package that has more than 100K lines of code and may lead to memory explosion. For example, CodeSurfer failed to generate PDGs for the chromium-browser package from Ubuntu, containing 12,826,166 lines of C/C++ code. This is the reason why the 101 packages mentioned in Section 5 failed. One solution is to extract individual modules out of these packages for compositional analysis. This is however non-trivial and we leave it as our future work.

Customized Certificate Validation in OpenSSL. SSLint models the API usage of SSL libraries through signatures, and then detects vulnerabilities in the usage through graph queries. However, instead of existing well-defined APIs, OpenSSL also provides an interface for developers to customize the certification validation process by a callback function. In particular, developers can specify a custom callback function that accepts the result of built-in verification and the X509 certificate and returns the developer's decision to accept or reject the certificate. As custom validation does not follow any existing API usage, our analysis cannot not model the behavior of such callbacks.

For this reason, we manually analyze all the callback functions that SSLint finds in 18 software packages. In all cases, we manually analyze the condition for each branch with a return instruction, and then decide whether the acceptance condition is vulnerable. For instance, if a custom validation allows self-signed or expired certificate, the manual analysis considers it as vulnerable.

Software Configurability. SSLint detects SSL vulnerabilities in applications, but not the intention of human beings. In practice, we find that some software has two branches for certificate validations: one is vulnerable and the other is secure. Then, the software gives the option to the user to select the branch. Such a practice is defined as software configurability, because a user can configure the software in her preferred way. SSLint successfully detects the vulnerable code that exists in the vulnerable branch of the certificate validation, however we are not going to argue whether this is indeed vulnerable, because the user is aware and has explicitly consented

to accept such insecurity. Examples of such software are "ftp-ssl" and "perdition" in Ubuntu 12.04.

It is worth noting that despite the above limitations, SSLINT is a capable auditing tool. As shown in this paper, it can be used to vet SSL usage in applications at scale and has already been applied to an entire operating system distribution resulting in the discovery of 29 previously unknown vulnerabilities.

7 Related Work

Vulnerabilities in SSL usage: A few works in the past have analyzed application vulnerabilities due to improper usage of SSL/TLS. Georgiev et al. [2] attempted MITM attacks against several applications and found over twenty certificate and hostname verification vulnerabilities. Their pioneering work shed light on a number of critical design flaws in the APIs of SSL libraries, and several vulnerabilities in middleware and applications. Their work is a natural starting point of our work. Their methodology involves black-box dynamic analysis involving setting up and testing the applications. Our approach has the goal of scaling the task of vulnerability analysis to hundreds of packages, something that cannot be done using their methodology because of the high setup cost. Our analysis approach is automated and scalable (we were able to analyze 492 software packages with no human effort).

Fahl et al. [3] and Sounthiraraj et al. [4] found SSL validation vulnerabilities in the Java code of Android applications. In Java the default SSL manager classes validate certificates/hostnames. Validation problems in Java may arise only when custom manager classes, i.e., custom validation code, are used. Both MalloDroid and SMV-Hunter identify such custom code and then use manual and automatic dynamic analysis respectively for vulnerability detection by exercising standard Android GUI interfaces. Thus there are two major differences between these two works and our work. First, validation by default is not the situation in the case of C/C++ SSL libraries and so we focus on correctness of SSL API usage. To achieve our goals, we modeled SSL API usage over control and data flow artifacts derived from a sophisticated static analysis; such techniques have not previously been used in the context of SSL validation. Second, the strategy of vulnerability detection by exercising standard GUI interfaces does not work for our applications such as mail servers and clients that do not share such common interfaces and require manual configuration to run.

Other SSL security works: Clark and Oorschot [5] present a comprehensive survey of SSL security. Several vulnerabilities have been found in SSL implementations and also in the protocols themselves. Examples include authentication vulnerabilities [6] and others such as Heartbleed [34], Debian OpenSSL predictable random numbers [35], and POODLE [36]. Our work is different from all these in that we find vulnerabilities in applications using SSL rather than the SSL implementations or specifications themselves. Security issues also arise due to certificate forgery, caused by cryptographic hash collisions [37] or CA compromise [38], [39]. Other attacks may exploit certificate validation quirks in different software [40]. Researchers have also studied SSL warnings in browsers [7], [8]. All these works and possible attacks are beyond the scope of this paper, which specifically targets SSL API usage in applications.

Vulnerability detection by static analysis: Static code analysis has been widely used to detect various vulnerabilities. Data flow vulnerabilities that compromise integrity such as cross site scripting and SQL injections are formulated as unsanitized data flow of untrusted input to a sink that should be protected [41]–[44]. Similarly, some vulnerabilities compromising confidentiality may be formulated as unsanitized data flow from a protected source to a public sink [45]. SSLINT applies similar techniques but for the purpose of detecting improper API usage. Like SSLINT, Egele et al. also use static analysis to check for vulnerabilities arising due to improper usage of cryptographic APIs in Android [46]. The scope of our work is different: we identify improper usage of SSL APIs and we did found several such vulnerabilities. Yamaguchi et al. [47] have modeled vulnerabilities as graph traversals on a combination of abstract syntax trees, control flow graphs, and program dependence graphs. While they detect vulnerabilities in Linux kernel, our work focuses on SSL usage vulnerabilities, which needed us to define signatures that are more expressive. Whereas their framework is more expressive than ours, we have found our approach based on program dependence graphs to suffice in detecting improper usage of SSL APIs.

Vulnerability signatures: Our signatures may be seen in light of past work on vulnerability signatures [14], [48]–[51] in intrusion detection. Such a signature is representative of the vulnerability itself and may be used to detect if a payload exploits the given vulnerability. Brumley et al. [15] explore the representation of vulnerability signatures in various classes, such as Turing machines, symbolic constraints, and regular expressions, and examine their precision. Instead of representing vulnerabilities, our signatures provide the exact representation for correct API usage. Our representation of signatures as queries on program dependence graphs is amenable to static analysis and allows us to be expressive enough to accurately model all SSL API usage cases.

8 Conclusion

Incorrect usage of a library implementing SSL/TLS protocols makes the software using the library vulnerable to man-in-the-middle (MITM) attacks. Finding such vulnerabilities statically is made challenging due to the data and control dependences interleaved in the API usage of different SSL libraries. In this paper, we present SSLINT, a static analysis tool that match a program dependence graph with a hand-crafted, precise signature modeling

the correct logic usage of SSL libraries. Because SSLINT matches the correct logic of library usage, any violations of the modeled behavior lead to a vulnerability. In practice, we made two signatures tailor made for popular C/C++ SSL libraries, namely OpenSSL and GnuTLS.

We have evaluated 492 software packages and identified 29 previously unknown vulnerabilities. Then, we reported our findings to developers of the software and received 14 confirmations, out of which, 5 have already fixed the vulnerability. For those we have not received a confirmation from, we perform a dynamic auditing to verify the found vulnerabilities, and the result shows that all of them are vulnerable to a MITM attack.

ACKNOWLEDGMENTS

This research was supported in part by the National Natural Science Foundation of China under Grant No. 61472209, by the U.S. National Science Foundation under Grants CNS-1408790, CNS-1065537, DGE-1069311 and by U.S. Defense Advanced Research Projects Agency under agreement number FA8750-12-C-0166.

REFERENCES

[1] "RFC 5246: The transport layer security (TLS) protocol version 1.2." https://datatracker.ietf.org/doc/rfc5246, 2008.
[2] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and Communications Security*. ACM, 2012, pp. 38–49.
[3] 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 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 50–61.
[4] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *Proceedings of the 19th Network and Distributed System Security Symposium. San Diego, California, USA*, 2014.
[5] J. Clark and P. C. van Oorschot, "Sok: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements." in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 511–525.
[6] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations." in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014.
[7] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer, "Here's my cert, so trust me, maybe?: understanding tls errors on the web," in *Proceedings of the 22nd international conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2013, pp. 59–70.
[8] D. Akhawe and A. P. Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness." in *Usenix Security*, 2013, pp. 257–272.
[9] "ScrollZ IRC client." http://www.scrollz.info/home.php.
[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.
[11] "Documents of OpenSSL library." https://www.openssl.org/docs/ssl/ssl.html.
[12] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23, pp. 2435–2463, 1999.
[13] The Snort Project, "Snort, the open-source network intrusion detection system." http://www.snort.org/.
[14] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, "Shield: Vulnerability-driven network filters for preventing known vulnerability e xploits," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 193–204, 2004.
[15] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 15–pp.
[16] P. T. Wood, "Query languages for graph databases." *ACM SIGMOD Record*, vol. 41, no. 1, pp. 50–60, 2012.
[17] GrammaTech Inc., "CodeSurfer®: Code Browser." http://www.grammatech.com/research/technologies/codesurfer.
[18] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.
[19] "The t. j. watson libraries for analysis (wala)," http://wala.sourceforge.net/wiki/index.php/Main_Page.
[20] "Wireshark." https://www.wireshark.org/.
[21] "Launchpad: a software collaboration platform." https://launchpad.net/.
[22] "RFC 2595: Using TLS with IMAP, POP3 and ACAP." https://datatracker.ietf.org/doc/rfc5246, 1999.
[23] "RFC 3207: SMTP Service Extension for Secure SMTP over Transport Layer Security." https://datatracker.ietf.org/doc/rfc3207, 2002.
[24] "Xfce4-Mailwatch-Plugin." http://goodies.xfce.org/projects/panel-plugins/xfce4-mailwatch-plugin.
[25] "Ubuntu popularity contest," http://popcon.ubuntu.com/, 2014.
[26] "Mailfilter: The Anti-Spam Utility." http://mailfilter.sourceforge.net/index.html.
[27] "Exim Internet Mailer." http://www.exim.org/.
[28] "RFC draft: SMTP security via opportunistic DANE TLS." https://datatracker.ietf.org/doc/draft-ietf-dane-smtp-with-dane, 2014.
[29] "DMA: DragonFly Mail Agent." https://github.com/corecode/dma/.
[30] "EPIC: Enhanced Programmable ircII Client." http://www.epicsol.org/.
[31] "The Prayer Webmail System." http://www-uxsup.csx.cam.ac.uk/~dpc22/prayer/.
[32] "FreeTDS." http://www.freetds.org/.
[33] "Squirrelmail's imap proxy," http://www.imapproxy.org/.
[34] "CVE-2014-0160," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.
[35] "CVE-2008-0166," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166.
[36] "CVE-2014-3566," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3566.
[37] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. De Weger, "Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate," in *Advances in Cryptology-CRYPTO 2009*. Springer, 2009, pp. 55–69.
[38] "Report of incident on 15-mar-2011," 2011, https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html.
[39] E. Mills, "Fraudulent google certificate points to internet attack," 2011, http://www.cnet.com/news/fraudulent-google-certificate-points-to-internet-attack/.
[40] D. Kaminsky, M. L. Patterson, and L. Sassaman, "Pki layer cake: new collision attacks against the global x. 509 infrastructure," in *Financial Cryptography and Data Security*. Springer, 2010, pp. 289–303.
[41] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis." in *Usenix Security*, 2005, pp. 18–18.
[42] X. Zhang, A. Edwards, and T. Jaeger, "Using cqual for static analysis of authorization hook placement." in *USENIX Security Symposium*, 2002, pp. 33–48.
[43] A. P. Sistla, V. Venkatakrishnan, M. Zhou, and H. Branske, "Cmv: Automatic verification of complete mediation for java virtual machines," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM, 2008, pp. 100–111.
[44] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov, "A security policy oracle: detecting security holes using multiple api implementations," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 343–354.
[45] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise

context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, p. 29.

[46] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 73–84.

[47] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs." in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014.

[48] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv, "Netshield: massive semantics-based vulnerability signature matching for high-speed networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 279–290, 2011.

[49] Y. Cao, X. Pan, Y. Chen, and J. Zhuge, "Jshield: towards real-time and vulnerability-based detection of polluted drive-by download attacks," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 466–475.

[50] L. Wang, Z. Li, Y. Chen, Z. Fu, and X. Li, "Thwarting zero-day polymorphic worms with network-level length-based signature generation," *ACM/IEEE Transaction on Networking*, vol. 18, no. 1, 2010.

[51] Z. Li, L. Wang, Y. Chen, and Z. Fu, "Network-based and attack-resilient length signature generation for zero-day polymorphic worms," in *Proc. of the 14th IEEE International Conference on Network Protocols (ICNP)*, 2007.