Analyzing Implementation-Based SSL/TLS Vulnerabilities with Binary Semantics Analysis

Li Wang[1](✉), Yi Yang[2], and Goutham Reddy Alavalapati[1]

[1] Fontbonne University, Clayton 63105, USA
{lwang,gralavalapati}@fontbonne.edu
[2] Northeastern Illinois University, Chicago 60625, USA
yyang@neiu.edu

Abstract. SSL/TLS are cryptographic protocols created to protect the security and privacy over computer network communication. As a critical security infrastructure on the internet, it has been widely used for decades in various network related applications, such as HTTPs, SMTPs, FTPs, and so on. Although it is designed to "protect" the network communication, it also has some security concerns. In this paper, we present the feasibility of analyzing implementation-based SSL/TLS vulnerabilities with binary semantics analysis. We use a basic-blocks-sequence based binary semantics comparison method to conduct vulnerability analysis on SSL/TLS vulnerabilities. We abstract a vulnerability execution trace as a "signature". By comparing the semantic similarity of a target program's execution trace and a vulnerability's "signature", we are able to detect whether the target program contains the vulnerability or not. We analyzed the well-known Heartbleed vulnerability and other implementation based vulnerabilities in representative network applications which use two popular SSL/TLS libraries, OpenSSL and mbedTLS. The evaluation result shows that our basic-blocks-sequence based binary semantics comparison method is effective on analyzing the existence of various implementation based SSL/TLS vulnerabilities.

Keywords: SSL/TLS vulnerability · Program vulnerability analysis · Binary semantics analysis · Binary similarity comparison · Symbolic execution · Dynamic analysis

1 Introduction

Secure Sockets Layer (SSL) and Transportation Layer Security (TLS) are cryptographic protocols created to protect the security and privacy over computer network communication. SSL and TLS are two different names of the cryptographic protocols during the different development phases. As a critical security infrastructure on the internet, SSL/TLS has been widely used for decades to protect various network related applications, such as HTTPs, SMTPs, FTPs,

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2025
Published by Springer Nature Switzerland AG 2025. All Rights Reserved
H. Duan et al. (Eds.): SecureComm 2023, LNICST 568, pp. 371–394, 2025.
https://doi.org/10.1007/978-3-031-64954-7_19

SSL Handshake	SSL Change Cipher Spec	SSL Alert	HTTP/FTP/SMTP...
SSL Record			
TCP/IP			

Fig. 1. SSL protocol structure

and so on. SSL/TLS protocols are integrated with four parts, Handshake protocol, Record protocol, Alert protocol, and Change Cipher Spec protocol, which work together to provide confidentiality, integrity, authenticity, and other related security services to the network communication. Figure 1 shows the SSL protocol structure. Among the four parts, the most important two parts are Handshake protocol and Record protocol. The Handshake protocol is in charge of negotiating the cryptographic parameters, such as cryptography algorithm suite, compression methods, random numbers, and so on. Through Handshake protocol, the client and server validate each other and prepare a secure communication channel for the later communications. Record protocol is located underneath the other three protocols, which is responsible for encrypting and decrypting the actual network traffic using the negotiated keys and cryptography algorithms prepared by Handshake protocol. It is the Record protocol to provide confidentiality and integrity security services to the protected network communication.

Although SSL/TLS is designed to "protect" the network communication, it also has some security concerns. Various security flaws have been accompanied with SSL/TLS protocols since it was developed from the first day. There are mainly two categories of vulnerabilities in SSL/TLS protocols. One is design based vulnerabilities, and the other is implementation-based vulnerabilities. The design based vulnerabilities are mainly caused due to the vulnerable design of SSL/TLS protocols. In 1998, Bleichenbacher et al. [4] discovered an adaptive chosen-ciphertext attack on SSL 3.0 protocol. This attack is rooted in RSA PKCS #1 encryption standard. By observing a large number of different error messages returned from an SSL server, an attacker is able to detect ciphertext where the plaintext started with 0x0020 through padding oracle attack. After trying 300 thousand to 2 million chosen ciphertexts choices, the attacker is able to guess the key information. Aviram et al. [1] continue Bleichenbacher's previous work and found the Decrypting RSA using Obsolete and Weakened eNcryption (DROWN) attack in 2016. DROWN attack still makes use of downgrade weakness of SSL protocol. With the help of DROWN attack, an attacker can manage to decrypt the session key even without the RSA private key. Unlike previous Bleichenbacher's attack, DROWN attack is able to finish the break with only one thousand times of guess, and it optimizes the chosen ciphertexts content to greatly improve the attacking efficiency. DROWN attack makes use of the weak cryptography algorithms supported in SSL 2.0. Although SSL 2.0 has been dep-

recated for many years, there are still millions of servers providing support of SSL 2.0. Consequently, over tens of millions of servers are affected by DROWN attack.

Besides, SSL/TLS protocols also contain implementation-based vulnerabilities. The implementation-based vulnerabilities, just as its name implies, is caused by the implementation of SSL/TLS protocols. The most well-known and influential implementation-based SSL/TLS vulnerability is OpenSSL Heartbleed [10]. Due to the heartbeat service exists in both communication sides, either server or client side using OpenSSL is subject to Heartbleed vulnerability. The vulnerable heartbeat extension implementation enables an unwanted memory leak because of failing to verify a user given payload length in a heartbeat request. Consider the payload length is two bytes, Heartbleed vulnerability allows a forged heartbeat request to ask up to 64K bytes content from the vulnerable side. The leak information may include email address, cryptographic secret keys, user credentials, and other secrets. We present the detail of Heartbeat vulnerability analysis in the evaluation section. There are many other implementation-based SSL/TLS vulnerabilities [35], which may cause typical SSL/TLS attacks, such as denial-of-service attack and man-in-the-middle attack. Besides, with the development of new vulnerability discovery technologies, such as dynamic taint analysis [8,43,51] and fuzzing [45,54], more and more implementation-based vulnerabilities are detected. In this work, we will focus on analyzing implementation based SSL/TLS vulnerabilities.

There are a lot of source code level vulnerability detection research [53,60,61]. However, source code is typical unavailable in many vulnerability detection scenarios. For example, when we are dealing with most commercial software or legacy systems, binary code is the only available code. Besides, the binary code may be stripped and does not contain symbol information, such as function names and data type. A vulnerability detection solution in stripped binary code is needed. We use basic-blocks-sequence based binary semantics comparison (for short, we use "basic-blocks-sequence semantics comparison" in the rest of the paper) to identify the semantics similarity of binaries. By comparing the execution traces of programs, basic-blocks-sequence semantics comparison can identify whether the programs are semantically equal or not. It can be used in various binary similarity research scenarios, such as patch detection [59], malware analysis and detection [38,57], software plagiarism detection [32,33], and so on. Consider basic-blocks-sequence semantics comparison only requires binary code (either stripped or unstriped), it could be a promising candidate solution on vulnerability detection in stripped binaries.

In this paper, we present the feasibility of analyzing implementation-based SSL/TLS vulnerabilities with basic-blocks-sequence semantics comparison. We abstract a vulnerability execution trace as a "signature" from vulnerable libraries. By comparing the semantics of a target program's execution trace and a vulnerability's "signature", we are able to detect whether the target program contains the vulnerability or not. To compare the semantics of program execution traces, basic-blocks-sequence semantics comparison models the similarity

comparison unit with basic blocks sequences instead of individual basic blocks. Compared with traditional binary semantics comparison schemes, basic-blocks-sequence semantics comparison is able to catch the code semantics beyond the boundary of a single basic block. Besides, it has an advantage of detecting code similarity over several obfuscation schemes, like loop unrolling, control-flow flattening, and so on. We analyzed the well-known Heartbleed vulnerability and other implementation-based vulnerabilities in two popular SSL/TLS libraries, OpenSSL [46] and mbedTLS [34]. For each library, we evaluated the representative network applications. The evaluation result shows that our scheme is effective on analyzing the existence of various implementation-based SSL/TLS vulnerabilities.

The rest of the paper is organized as follows. Section 2 introduces the background and motivation of our work. Section 3 and 4 give the design and implementation details on how we can use basic-blocks-sequence semantics comparison to analyze and evaluate the implementation-based SSL/TLS vulnerabilities. The evaluation of vulnerability analysis work is presented in Sect. 5, and Sect. 6 discusses some limitations of our work, such as our method may fall in the limitations of the dynamic analysis method. Last, the related work and conclusion are given in Sect. 7 and 8.

2 Background

In this section, we first give the current security situation of software industry. Then, we introduce the existing vulnerabilities analysis work of SSL/TLS protocols. Last, we present the basis of program semantics comparison and bring up our basic-blocks-sequence semantics comparison method.

Software vulnerability is an inevitable threat to modern computer systems. According to National Vulnerability Database [44], there are total of 20171 vulnerabilities reported in 2021. However, this number has increased to 25227 in 2022, which is about 25% increase over 2021. There are 3691 Common Vulnerabilities and Exposures (CVE) records related to "SSL" keyword in National Vulnerability Database. The high number of published CVE records show the necessity of security analysis over SSL/TLS design and implementation. A lot of work has been done on SSL/TLS security analysis [2,11,19,22,35]. In 1996, Wagner et al. [56] brought the first systematic security analysis work on SSL 3.0. The authors analyzed the confidentiality and authenticity over SSL 3.0 and presented several security flaws. In 2014, Meyer et al. [35] presented an overview of SSL/TLS attacks over the past 17 years. He summarized both theoretical and practical attacks on all four SSL/TLS protocol components and gave lessons learned for each type of SSL/TLS protocol attacks. Although the existing work provides security evidence on evaluating SSL/TLS security properties, most of them are explanatory and prone to design based SSL/TLS vulnerabilities. In this paper, we use basic-blocks-sequence semantics comparison method to analyze the implementation-based SSL/TLS vulnerabilities.

Binary semantics comparison compares the semantics difference of programs at binary level. It has been widely used in many security research scenarios, such

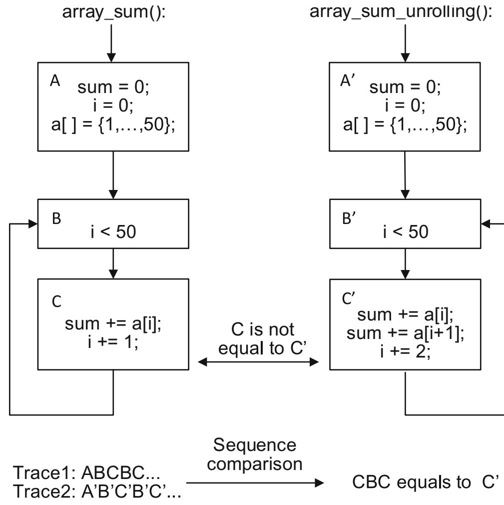

Fig. 2. Semantics equivalence caught with basic-blocks-sequence semantics in loop unrolling.

as malware detection, code clone detection, software plagiarism detection, and so on. Due to the significance of binary semantics comparison, a lot of research has been done. BinHunt series work [13,37] first leverages symbolic execution and constraint solving to match the binary semantics in basic blocks. CoP [32] is a binary semantics based software plagiarism detection tool, which applies the longest common subsequence algorithm in block-centric binary semantics comparison. Similarly, malware analysts also utilize semantics-based [38,57] binary comparison to identify malware lineage and analyze malware behaviors. However, most of the existing semantics comparison methods are "block-centric", which models the similarity comparison unit with individual basic blocks. The block-centric methods are effective in catching code semantics and able to deal with obfuscation schemes within individual basic blocks, but they suffer a common limitation that they cannot detect the program semantics beyond the boundary of a single basic block.

As a result, basic-blocks-sequence semantics comparison method is proposed to address the limitation of "block-centric" binary semantics comparison in most existing binary semantics comparison methods. Basic-blocks-sequence semantics comparison is capable of capturing the semantics cross individual basic blocks. Figure 2 gives an example of how basic-blocks-sequence semantics comparison is able to capture the semantics cross individual basic blocks. Control flow graphs are given for two functions, *array_sum()* and *array_sum_unrolling()*. In the figure, each node marked with a capitalized letter represents a basic block. We can see both functions are executing a *for* loop to calculate a sum of integer range [0, 50). Compared with *array_sum()*, the *for* loop in *array_sum_unrolling()* is optimized with loop unrolling. As a result, the block C' of *array_sum_unrolling()* does not

equal to block C in *array_sum()* for C' has a different number of *add* operations. If we use "block-centric" semantics comparison method to compare the two functions' semantics, we will probably draw a conclusion that these two functions are not semantically equal because block C and block C' are not equal. However, the fact is the function *array_sum()* and *array_sum_unrolling()* are semantically equal. If we use basic-blocks-sequence semantics comparison, which is able to identify the block sequence CBC is equal to C', we may conclude that these two functions are equal. The example in Fig. 2 shows the strength of basic-blocks-sequence semantics comparison. More details about our basic-blocks-sequence semantics comparison method will be given in the next section.

3 Methodology

In this section, we introduce the design details of vulnerability analysis with basic-blocks-sequence based semantics comparison. We first give an outline of our method. Then, we introduce how we retrieve a vulnerability's signature trace from a vulnerable program's execution trace. Last, we present semantics comparison basis and bring details of basic-blocks-sequence semantics comparison.

3.1 Outline

Figure 3 gives an overview of our method. As can be seen in the figure, the dash area represents the vulnerable part in source code and binary code. Following the figure, the vulnerable source code will be first compiled into vulnerable binary code. Then, the vulnerable binary code will be executed, and its execution trace will be collected during runtime. The execution trace is collected with Pin [30], an Intel dynamic instrumentation tool. Then, we retrieve a "signature" trace from the collected execution trace, and the "signature" trace contains the trace of the vulnerable part of the program. Last, we compare the semantics of the "signature" trace with a target program's execution trace (we call it target trace) using basic-blocks-sequence semantics comparison. Based on the semantics comparison result, we are able to conclude whether the target program contains the vulnerability or not. In the next subsections, we introduce how we retrieve a vulnerability's signature trace from an execution trace.

3.2 Signature Trace Retrieval

As Fig. 3 shows, a "signature" trace is retrieved from the execution trace of the vulnerable program. Given the source code of a vulnerable program, we first find out the vulnerability site, which usually resides in a vulnerable function. The vulnerability may cover multiple lines of code in that function. We choose a part of the vulnerability as a signature and record its source code line numbers. Since the minimal comparison unit of basic-blocks-sequence semantics comparison is basic block, we will make sure the signature will not brake a basic block's

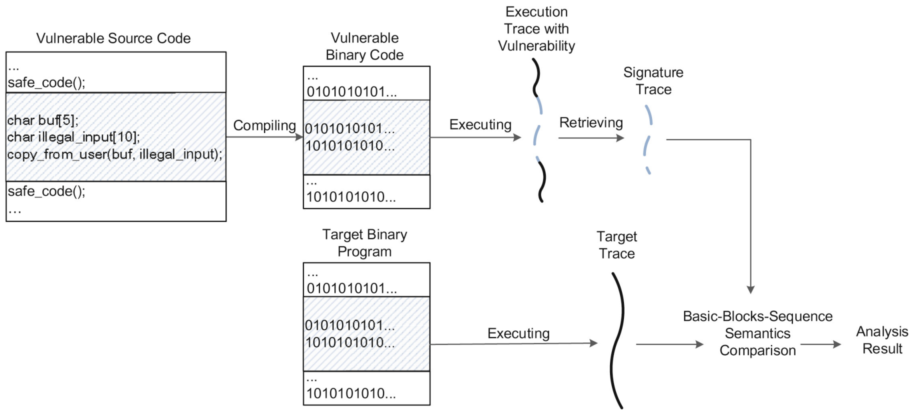

Fig. 3. Overview: Vulnerability analysis with basic-blocks-sequence semantics comparison

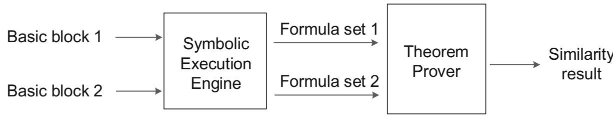

Fig. 4. Basic block semantics comparison.

boundary. Then we compile the vulnerable source code with debug option. The debug option will keep the code line number information in the compiled binary code, which can help us to locate the signature's corresponding instructions in the binary. Then, we locate the instructions of the signature in the binary code by using debug information. The signature instructions' position will be recorded as the offset of the vulnerable function. Then, we execute the vulnerable binary code and use Intel Pin [30] to collect an execution trace based on the offset of the vulnerable function. When executing the vulnerable program, we use proper input to activate the vulnerability in the program, which makes sure the vulnerability will be executed during running. As a result, the collected trace will be the execution trace of the vulnerable part we choose from the vulnerable function. We use the collected trace as the "signature" trace of the vulnerability. The "signature" trace will be used to compare with a target trace in the later basic-blocks-sequence semantics comparison phase.

3.3 Basic-Blocks-Sequence Based Semantics Comparison

Existing binary semantics comparison schemes compares the semantics difference of programs at binary level. The basic idea behind binary semantics comparison is a binary's semantics is interpreted by symbolically executing its instructions. In a typical binary semantics comparison scenario, a binary will be first divided

into basic blocks [13,32]. Each basic block will be symbolically executed with its input variables, such as registers or memory cells, and produce a set of symbol formulas as output. The generated symbol formulas include the arithmetic relations of the involved input variables and represent the semantics of that basic block. The similarity of two basic blocks is decided by the equivalence of their formula sets. To decide the equivalence of two formula sets, a pair-wise formula comparison is calculated by a theorem prover. The theorem prover conducts a constraint solving on two symbolic formulas and decides whether they are equivalent or not. If there is a pair-wise match between two symbol formula sets, the two basic blocks are regarded as semantically equivalent. Figure 4 shows the comparison process of two basic blocks. A limitation of the exiting binary semantics comparison methods is they are all "block-centric", which means the semantics similarity is calculated based on individual basic block's semantics similarity result. As a result, the "block-centric" binary semantics comparison methods may fail to catch the semantics cross the boundary of a single basic block (see Fig. 2). To address the limitations of "block-centric" binary semantics comparison methods, basic-blocks-sequence semantics comparison was proposed to extend the minimal semantics comparison unit to basic blocks sequences.

Basic blocks based binary semantics comparison is a binary semantics comparison method based on single or multiple basic blocks. This idea is originally comes from computational linguistics area [5]. Compared with existing binary semantics comparison methods, it is proposed to provide a fine-grained semantics comparison for all semantics-based binary semantics comparison research. By modeling the comparison unit with multiple basic blocks instead of a single basic block, basic-blocks-sequence based semantics comparison extends the semantics comparison unit to multiple basic blocks. Our method uses dynamic analysis to generate execution traces of a binary. To collect multiple basic blocks, it cuts an execution trace into different sizes of basic blocks sequences with a sliding window, and the length of the sliding window is by the number of basic blocks. By setting the sliding window length to 1, 2, 3, ..., N, the basic blocks sequences are collected. The collected basic block sequences will be transformed to formulas with symbolic execution, and the semantics comparison result is calculated similarly.

We apply basic-blocks-sequence based semantics comparison method to vulnerability analysis. To measure the vulnerability analysis result, we define a match score for the compared "signature" trace and the target trace. The match score indicates whether the vulnerability exists in the target trace, and it is defined as follows (bb stands for basic block):

$$\text{MatchScore} = \frac{\text{Number of matched signature bb elements}}{\text{Number of total signature bb elements}}$$

For example, we have a signature trace s1: 12345 and t1: 123467. Each number position represents one basic block, and the number value represents the content of the basic block. s1's basic block set s1_bb_set: {1,2,3,4,5} has five elements, while t1's basic block set t1_bb_set: {1,2,3,4,6,7} has six elements. We collect basic-blocks-sequence (n = 1, each basic block sequence

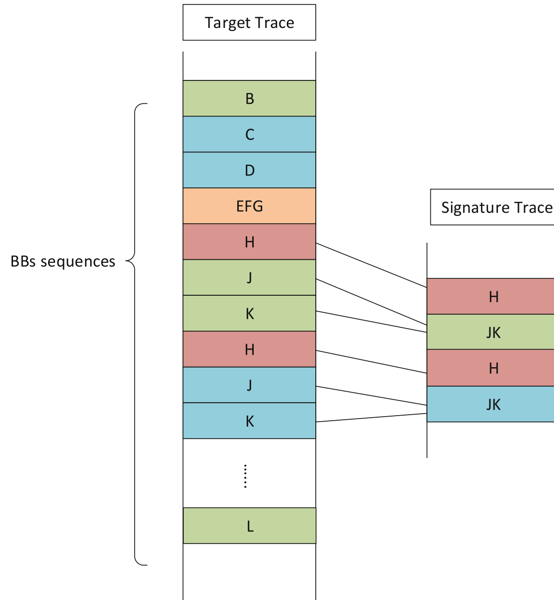

Fig. 5. Signature trace matching.

has 1 basic block) basic block sequences set s1_seqs_set: {1,2,3,4,5} for s1 and t1_seqs_set: {1,2,3,4,6,7} for t1 and make a pair-wise comparison of s1_seqs_set and t1_seqs_set. We can find pair-wise matches on 1-1, 2-2, 3-3, and 4-4, and the unmatched basic block sequences are 5, 6, and 7. Then we calculate the number of matched basic block elements in s1's basic block set s1_bb_set, which is 4. Give the fact that s1_bb_set has five elements, the match score for s1 and t1 is $4/5 = 0.8$. Figure 5 shows an example of signature trace matching. As Fig. 5 shows, each rectangle represents one basic blocks sequence. The capitalized letter stands for the sequence content, and the connected rectangles in the same color mean the matched equivalent basic-blocks sequences. Our definition measures the ratio of the number of matched basic block elements to the total number of basic block elements in a signature trace, which indicates the semantics relationship between the signature trace and the target trace. When comparing a vulnerability's signature trace with a program's execution trace, if the calculated match score is 1, we can conclude the program contains the vulnerability. Otherwise, we are unable to know whether the program contains the vulnerability or not.

4 Implementation

We developed a basic-blocks-sequence semantic comparison vulnerability analysis prototype with a 5k lines of code (C/C++ and Python). The prototype is composed of three parts, trace collection tools (Pin tool), symbolic execution

engine, and semantics comparison engine. We implemented different Pin tools to collect signature traces and target traces. The signature trace is collected with the vulnerable function name and signature offsets, while the target trace is collected by a target program's socket system call sequence. All the trace collection tools are implemented with Intel Pin [30]. More details about trace collection will be discussed in the next section.

5 Analyzing SSL/TLS Vulnerabilities with Basic-Blocks-Sequence Semantics Comparison

In this section, we analyze implementation-based SSL/TLS vulnerabilities with basic-blocks-sequence semantics comparison. We first give a detailed analysis on the OpenSSL Heartbleed vulnerability, for it is the most well-known and consequential implementation-based SSL/TLS vulnerability in the past ten years. Then, we evaluate some other typical implementation-based SSL/TLS vulnerabilities (OpenSSL) and the latest TLS side channel vulnerabilities (mbedTLS). Our evaluation covers various network applications on both the client side and server side, and the evaluation result shows that our basic-blocks-sequence semantics comparison method is effective on analyzing the existence of various implementation-based SSL/TLS vulnerabilities.

5.1 Testbed

We conduct the vulnerability analysis work on an Ubuntu 14.01 virtual machine. Our vulnerability security analysis work is based on two popular SSL/TLS implementations, OpenSSL and mbedTLS. We use OpenSSL-1.0.1f and MbedTLS-2.2.1, MbedLTS-2.13.0, MbedTLS-2.14.1, and MbedTLS-2.16.0 as vulnerable libraries. To evaluate the false positive of our method, we use OpenSSL-1.0.1u and MbedTLS-2.16.4 as patched libraries for the invulnerable applications. The evaluated OpenSSL applications cover curl-7.36.0, wget-1.15, mini_httpd-1.30, and nginx-1.4.7, and the MbedTLS applications include monkey-1.6.9, hiawatha-10.8.3, hiawatha-10.9, hiawatha-10.10, and openvpn-2.4.7. We collect the corresponding target traces and compare them with the "signature" traces. More details will be given in the following content.

5.2 Trace Collection

The collected trace includes instruction content, register values, CPU flag information, and memory cell information. We collect two kinds of traces, "signature" traces and target traces. For "signature" traces, we use the vulnerable function name and instruction offsets to collect it directly from the vulnerable libraries. The "signature" trace is only part of the vulnerability, which usually contains several basic blocks. For target traces, we use socket system calls instead of function names to identify which part of the application we want to record. Consider

```
unsigned int payload;
unsigned int padding = 16; /* Use minimum padding */

hbtype = *p++;
n2s(p, payload); /* Read payload from user */
p1 = p;
[...]
/* Allocate memory for the response, here fail to
   check payload value */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
[...]
/* Copy unchecked length of memory to buffer */
memcpy(bp, pl, payload);
```

Fig. 6. OpenSSL Heartbleed Vulnerability

```
/* Read type and payload length first. If payload
   length is 0, return 0 */
if (1 + 2 + 16 > s->s3->rrec.length)
    return 0;
hbtype = *p++;
n2s(p, payload);

/* Check if payload length > actual payload */
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0;
p1 = p;
```

Fig. 7. OpenSSL Heartbleed Patch

different applications may use different socket system calls to send/receive packages, we wrote different pin tools for different network applications. For a server application, the SSL/TLS related execution usually happens after an accept() system call (cases may vary). For a client application, the SSL/TLS related execution probably happened after a connect() system call (cases may vary). Our pin tools follow the socket system call sequence to collect traces. Because we use system call sequence to collect target traces, the target traces may contain a lot of unrelated instructions, such as unrelated libraries. We use pin tool to find out the range of the unrelated libraries, and those library instructions will be filtered out before the target trace is compared with signature traces.

5.3 Heartbleed Analysis

The Heartbleed vulnerability CVE-2014-0160 was first published on April 7th, 2014. There was an estimate that about 22–55% HTTPs services on top 100 websites and over 40 mainstream server products were affected [10]. Besides, Android devices, Bitcoin client, Tor network, and wireless network were also impacted by the Heartbleed vulnerability. We analyze the Heartbleed vulnerability on both server and client side respectively.

Figures 6 and 7 show the Heartbleed vulnerability and the patch code. We analyze the vulnerability and patch code to explain how the Heartbleed memory leak happens in real communication. As Fig. 6 shows, the vulnerable code first reads the heartbeat type and payload length information (here the payload length is given directly by user). Then, a buffer is allocated with a length

```
Listening on :4433 for tls clients
Connection from: 192.168.0.4:57548
Client returned 65517 (0xffed) bytes
0000: 2d 03 03 52 34 c6 6d 86 8d e8 40 97 da ee 7e 21  -..R4.m...@...~!
0010: c4 1d 2e 9f e9 60 5f 05 b0 ce af 7e b7 95 8c 33  .....`_....~...3
0020: 42 3f d5 00 c0 30 00 00 05 00 0f 00 01 01 00 00  B?...0..........
0030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
4540: 00 b9 44 00 00 00 00 00 00 00 00 00 00 00 00 00  ..D.............
89f0: 00 00 00 00 00 00 00 00 00 a9 01 00 00 01 00 01  ................
8a00: 36 03 03 7a 88 72 14 44 d0 65 96 27 18 b4 c2 c3  6..z.r.D.e.'....
8a10: 33 d9 a8 08 b1 9f f5 49 e0 b5 bf 9a f1 da 89 b1  3......I........
8a20: 04 be 06 00 00 a0 c0 30 c0 2c c0 28 c0 24 c0 14  .......0.,.(.$..
8a30: c0 0a c0 22 c0 21 00 a3 00 9f 00 6b 00 6a 00 39  ...".!.....k.j.9
8a40: 00 38 00 88 00 87 c0 32 c0 2e c0 2a c0 26 c0 0f  .8.....2...*.&..
8a50: c0 05 00 9d 00 3d 00 35 00 84 c0 12 c0 08 c0 1c  .....=.5........
8a60: c0 1b 00 16 00 13 c0 0d c0 03 00 0a c0 2f c0 2b  ............./.+
8a70: c0 27 c0 23 c0 13 c0 09 c0 1f c0 1e 00 a2 00 9e  .'.#............
8a80: 00 67 00 40 00 33 00 32 00 9a 00 99 00 45 00 44  .g.@.3.2.....E.D
8a90: c0 31 c0 2d c0 29 c0 25 c0 0e c0 04 00 9c 00 3c  .1.-.).%.......<
8aa0: 00 2f 00 96 00 41 00 07 c0 11 c0 07 c0 0c c0 02  ./...A..........
8ab0: 00 05 00 04 00 15 00 12 00 09 00 14 00 11 00 08  ................
8ac0: 00 06 00 03 00 ff 01 00 00 6d 00 0b 00 04 03 00  .........m......
8ad0: 01 02 00 0a 00 34 00 32 00 0e 00 0d 00 19 00 0b  .....4.2........
8ae0: 00 0c 00 18 00 09 00 0a 00 16 00 17 00 08 00 06  ................
8af0: 00 07 00 14 00 15 00 04 00 05 00 12 00 13 00 01  ................
8b00: 00 02 00 03 00 0f 00 10 00 11 00 23 00 00 00 0d  ...........#....
8b10: 00 20 00 1e 06 01 06 02 06 03 05 01 05 02 05 03  . ..............
8b20: 04 01 04 02 04 03 03 01 03 02 03 03 02 01 02 02  ................
8b30: 02 03 00 0f 00 01 01 02 00 00 2d 03 03 52 34 c6  ..........-..R4.
8b40: 6d 86 8d e8 40 97 da ee 7e 21 c4 1d 2e 9f e9 60  m...@...~!.....`
8b50: 5f 05 b0 ce af 7e b7 95 8c 33 42 3f d5 00 c0 30  _....~...3B?...0
8b60: 00 00 05 00 0f 00 01 01 00 00 00 00 00 00 00 00  ................
8ba0: 00 09 10 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
9ba0: 00 00 00 00 00 00 00 00 00 09 10 00 00 00 00 00  ................
abb0: 00 f9 00 00 00 03 03 00 00 00 00 00 00 00 00 00  ................
ac50: 00 00 00 00 00 00 00 00 00 01 00 00 00 2c 01 00  .............,..
ac60: 00 c8 b8 a7 5e 00 00 00 00 00 00 00 00 00 00 00  ....^...........
aca0: 00 00 00 00 00 00 00 00 00 09 00 01 00 02 ff ed  ................
acb0: 2d 03 03 52 34 c6 6d 86 8d e8 40 97 da ee 7e 21  -..R4.m...@...~!
acc0: c4 1d 2e 9f e9 60 5f 05 b0 ce af 7e b7 95 8c 33  .....`_....~...3
acd0: 42 3f d5 00 c0 30 00 00 05 00 0f 00 01 01 00 00  B?...0..........
f1f0: 00 b9 44 00 00 00 00 00 00 00 00 00 00 00 00 00  ..D.............
```

Fig. 8. Client side Heartbleed memory leak

of 1 + 2 + payload + padding. Notice the buffer length is allocated using the user provided payload length, which is unverified and could be illegal. When the memory copy happens in memcpy(bp, pl, payload) function, the vulnerable code will copy the memory content to the buffer with user provided (possibly illegal) payload length. As a result, a malicious user can indicate any payload length value of two bytes to request memory content from the vulnerable server. Comparatively, Fig. 7 shows the patch code. We can see in the patch code, the patched code first checks whether the actual payload length is 0 ($1 + 2 + 16 > s$ -> s3 -> rrec.length). If the payload length is 0, the function just returns. Before allocating the buffer, the patched code identifies whether the given payload length is larger than the actual payload length. If yes, no response happens and the function returns. In the patch code, if the given payload length is 0 or larger the actual payload length, the function just returns and no memory leak happens. As we can see in the analysis, the Heartbleed vulnerability is caused by failing to verify user inputs, which leads to an unexpected memory leak.

We first analyze the Heartbleed vulnerability at the client side. Figure 8 shows the client side Heartbleed memory leak. The malicious server is configured to wait at 4433 port for the coming TLS connections. We control the malicious server to send a “forged” heartbeat request to the connected client and retrieve arbitrary length of memory content (up to 64KB) from the client side. As can be seen in the Fig. 8, the malicious server successfully retrieves 65517 bytes of

Fig. 9. Server side Heartbleed memory leak

memory content from the client. We use curl-7.36 and wget-1.15 with vulnerable OpenSSL-1.0.1f as vulnerable clients to collect target traces. Since client uses connect() system call to start an SSL/TLS connection, our pin tool starts collecting the execution traces after connect() system call. We compare the collected target traces with the heartbleed signature trace and the match score is 1, which shows the target traces are vulnerable to Heartbleed vulnerability [18]. To measure false positive, we compare the heartbleed signature trace with the target traces of curl and wget using patched version OpenSSL-1.0.1u, and the result is 0. The result 0 means the patched target traces are not vulnerable to Heartbleed vulnerability.

We also analyze server side Heartbleed vulnerability. Figure 9 shows the server side Heartbleed memory leak. We use mini_httpd-1.30 [47] with vulnerable OpenSSL-1.0.1f as vulnerable server. We set up a malicious client to send a "forged" heartbeat request to activate the memory leak. As can be seen in Fig. 9, the malicious user successfully received 65535 bytes of response including memory leak from mini_http server. In this example, the memory leak includes the service name, certificate information, and other sensitive information. Compared with the client side memory leak, we can see the server side Heartbleed vulnerability may lead to a higher chance of exposing sensitive data or secrets. We collect the target traces for mini_httpd and nginx using vulnerable OpenSSL-1.0.1f and compare it with the heartbleed signature trace. The match score is 1 showing both applications are subject to Heartbleed attack. Comparatively, we evaluate the mini_httpd and nginx programs using patched version OpenSSL-1.0.1u. The result score is 0, which means the two applications using a patched library does not have a heartbleed vulnerability.

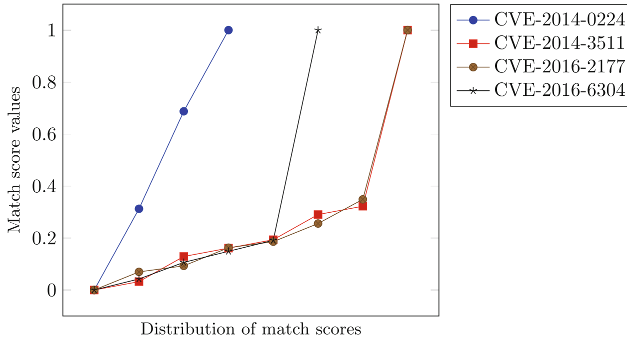

Fig. 10. Match scores for vulnerable server target traces

5.4 Other Typical SSL/TLS Vulnerabilities

We conduct the vulnerability analysis on other typical implementation-based SSL/TLS vulnerabilities, which are CVE-2014-0224, CVE-2014-3511, CVE-2016-2177, and CVE-2016-6304. These vulnerabilities are all at server side and caused by improper implementations, such as null pointer dereference, integer overflow, and so on. They may result in typical SSL/TLS attacks, such as denial-of-service attack, downgrade encryption algorithm, and man-in-the-middle attack. We evaluate two applications, mini_httpd-1.30 and nginx-1.4.7. For each application, we compile it with vulnerable SSL/TLS library and invulnerable (patched) SSL/TLS library and collect their running traces. As a result, we collect 8 target traces for each application, and each vulnerability per application has two target traces. We compare the signature and target trace pairs and calculate the match scores S, $S = F_{comp}(signature\ trace,\ target\ trace,\ n)$, *$n$ is the size of basic-blocks-sequence*. We have $n = 1, 2, 3, 4$ for calculating match scores. Consequently, for each vulnerability, we calculate 8 match scores. Figures 10 and Fig. 11 demonstrate the evaluation results for the two applications with and without SSL/TLS vulnerable library.

In the two figures, the y-axis stands for the match scores ranging from 0 to 1, and the x-axis stands for the possible number of different match score values (Note we calculate 8 match scores for each vulnerability. But there may have duplicates values among the 8 scores. E.g., when having $n = 2, 3$, the same signature and target trace pair may generate same match score. That explains why CVE-2014-0224 only has four score values in Fig. 10). We can see the scores distribution in two figures. In Fig. 10, the max match scores is 1, which means the vulnerabilities are detected in vulnerable target traces. Compared with Fig. 10, we can see the max match score in Fig. 11 is around 0.5, which means the vulnerabilities are not detected in invulnerable target traces. The vulnerability information and more explanation are as follows:

CVE-2014-0224: This vulnerability can be used to force the client and server to use vulnerable key material in SSL/TLS. As a result, the communication two sides may suffer a man-in-the-middle attack. Taking advantage of this vul-

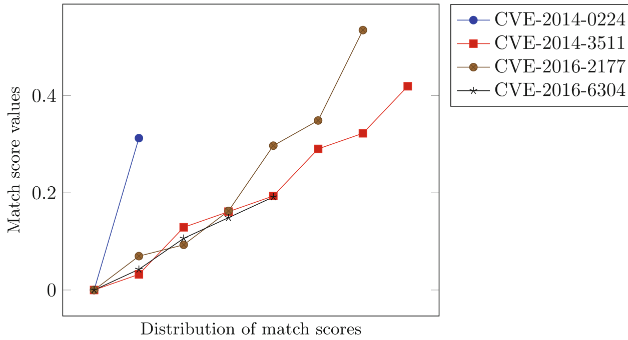

Fig. 11. Match scores for invulnerable server target traces

nerability, an attacker is able to decrypt and modify the network traffic during the communication. We compare the target traces with the signature trace, and the match scores result is 1, which shows these two server applications can be attacked with this vulnerability. For the target traces using patched library OpenSSL-1.0.1u, our evaluation scores (from 0 to 0.31) show there is no signature match.

CVE-2014-3511: By making use of it, an attacker is able to initialize a man-in-the-middle attack by downgrading both communication sides to use TLS 1.0, even if the client and server support a later TLS version. This vulnerability is caused by an improper implementation of the protocol. The target traces are compared with the signature traces and the result match score is 1. Both servers are subject to this man-in-the-middle attack. Similarly, the match score on target traces with patched library OpenSSL-1.0.1u is below 0.32.

CVE-2016-2177: An attacker can take advantage of this vulnerability to start a remote attack to cause a denial-of-service attack. The vulnerability is caused by an integer overflow, which makes the application crashed with a trigger input. We compare the target traces with the signature trace, and the match score shows this vulnerability exists in the two server applications. For the target traces using patched library OpenSSL-1.0.1u, there is no match score equal to 1.0.

CVE-2016-6304: With this vulnerability, a malicious client is able to send multiple illegal OCSP status request extensions. By keep sending large OCSP status request extensions, the client is able to cause the server an unbounded memory growth. As a result, the server will exhaust its memory and ends with a denial-of-service attack. Similarly, the comparison result of the target traces and the signature trace shows both two servers are vulnerable, and the match score is 1. Comparatively, the match score for target traces with patched library OpenSSL-1.0.1u is below 0.2.

The above result demonstrates analyzing typical SSL/TLS implementation based vulnerabilities with basic-blocks-sequence semantics comparison. However, with the development of vulnerability discovery and new attacking technologies, more and more new SSL/TLS vulnerabilities are reported, such as various side

channel based vulnerabilities. In the next subsection, we will analyze the recent side channel based SSL/TLS vulnerabilities.

5.5 Side Channel SSL/TLS Vulnerabilities

We also analyze the side channel SSL/TLS attack, which makes use of the information based on the implementation of computer systems to complete an attack. A lot of researchers have reported new side channel attacks on SSL/TLS [7,12,36,41,49,64]. Using side channel attacks, researchers [36,49] are able to realize the previously "solved" Bleichenbacher attack [4] based on the current SSL/TLS implementations. We use basic-blocks-sequence semantics comparison to analyze the side channel based SSL/TLS vulnerabilities. We evaluate monkey-1.6.9, hiawatha-10.8.3, hiawatha-10.9, and openvpn-2.4.7 with vulnerable libraries for the recent side channel vulnerabilities. Comparatively, we collect target traces of hiawatha-10.10 with patched library MbedTLS-2.16.4 for false positive tests. The collected target traces with vulnerable and patched libraries are compared with signature traces and the match scores are calculated. The vulnerability information and evaluation results are as follows:

CVE-2018-19608: Although the previous Bleichenbacher attack has been "solved" with padding oracle countermeasures, an attacker can still utilize timing variation and memory access variation to realize a Bleichenbacher attack. This vulnerability allows the attacker to make use of the timing side channel. The attacker can manage to downgrade the affected servers and recover all the 2048 bits of RSA plaintext in under 30 s [49], and the attacked servers lose the protection of RSA encryption. We evaluate this vulnerability with monkey-1.6.9 and hiawatha-10.8.3. The match scores of comparing the target traces and the signature trace are 1, which shows both servers are vulnerable. To measure the false positive, we measure the target trace of hiawatha-10.10 with patched library MbedTLS-2.16.4, and the result match score is 0.

CVE-2019-16910: This vulnerability is not caused by a timing side channel. Instead, It is caused by using a random number generator with insufficient entropy for blinding. The deterministic ECDSA calculation reuses the HMAC-DRBG to implement blinding. An attacker can make use of it and ask a victim to sign the same message many times. As a result, the private key may be recovered via a side channel attack. We evaluate this vulnerability on hiawatha-10.9 and openvpn-2.4.7 with an ECDSA certificate. The calculated match scores for both servers are 1, and the result shows the vulnerability exists in the collected target traces. For the target trace of hiawatha-10.10 with patched library MbedTLS-2.16.4, the evaluation result is 0.

CVE-2019-18222: This vulnerability is rooted in ECDSA. Due to the vulnerable bignum implementation is not with constant time and constant trace, it can be utilized to start a side channel attack. A local attacker is able to retrieve the blinded value, factor it, and recover the private key by brute force. The target traces are on hiawatha-10.9 and openvpn-2.4.7 with an ECDSA certificate. The match scores of comparing the target traces and the signature trace are 1, which shows this vulnerability exists in the two servers. Similarly, match

score 0 is reported for the target trace of hiawatha-10.10 with patched library MbedTLS-2.16.4.

Given the analysis result of the latest side channel vulnerabilities, we can see basic-blocks-sequence semantics comparison is able to analyze and identify the latest side channel vulnerabilities in popular SSL/TLS implementations (MbedTLS library). Combining the evaluation result of above two subsections, we can see our basic-blocks-sequence semantics comparison method is effective in identifying the vulnerability execution trace in the target traces. The performance of basic-blocks-sequence semantics comparison is discussed as follows.

5.6 Performance

Basic-blocks-sequence semantics comparison relies on comparing the basic blocks sequences of an execution trace. Usually, a longer execution trace will contain more basic blocks sequences and need more time to finish semantics comparison. Besides, the length of the symbolic formula will also affect the semantics comparison time. A formula with more input variables and calculations requires more comparison time. Consider the sizes of the collected target traces are quite different (from megabytes to gigabytes), we cut the target traces into small chunks to assist the following semantics comparing. By utilizing system call sequence information, we locate the possible position of the vulnerability trace and start the semantics comparison with nearby chunks. If there is a chunk in the target trace whose match score is 1, we conclude the vulnerability exists in the target trace. Also, another factor that may affect the performance of our method is the functions with complicated input-output dependencies [6]. These functions (like pseudo-random functions) will generate long symbolic formulas and make the semantics comparing time-consuming. We identify these functions with existing heuristics [15] and filter them out before conducting semantics comparison. Table 1 shows the statistic comparison time of target and signature traces used in our experiment, and the comparison is conducted on a 64bit Ubuntu 16.04 with Intel i7-11850 processor (2.5 GHz x 8) and 64 GB memory. The first row shows the number of instructions in the signature traces of different sizes (range from 16 to 47). For each signature trace, we use the number of target traces and total instruction numbers of target traces to calculate an average instruction number of the target traces. The second row in Table 1 shows the number of instructions of the target traces. In the second row, we can see the number of instructions of target traces is from 2812 to 3333. There is no ready answer to decide the size of the trace for different scenarios may have different requirements. We choose the size of target trace around 3000 instructions (from 2800 to 3400) heuristically based on a performance consideration. The comparison time of the signature trace and the target traces are accumulated, in order to calculate an average comparison time for that signature trace. The average comparison time for each signature trace is shown in the third row. We can see the average comparison time for different signature traces and target traces are different. The average comparison time for a signature of 16 instructions is 1.976 s (with target traces of 3079 instructions). When the signature trace has 39 instructions, the

average comparison time increase to 2.071 s with target traces of 2812 instructions. As can be seen in the table, both signature and target trace sizes affect the average comparison time. When the average target trace size is equal (3130 instructions), the longer signature trace usually requires more comparison time. Table 1 shows the data of this work, and it is for a reference purpose. Consider different vulnerabilities may generate different traces, and different traces may require different semantics comparison time. The semantic comparison time for target and signature traces in different vulnerabilities could be different.

Table 1. Target/Signature trace comparison time

Signature-trace size (# inst.)	16	30	37	39	43	47
Avg. target-trace size (# inst.)	3079	3333	3239	2812	3130	3130
Avg. comparison time (s)	1.976	3.876	3.160	2.071	2.108	2.449

6 Discussion

In this section, we discuss some potential limitations of our method. Basic-blocks-sequence semantics comparison makes use of dynamic execution traces to retrieve the programs' semantics information. As a result, it falls into the limitations of the dynamic analysis methods. Dynamic analysis methods take advantage of analysis accuracy because it is based on a program's running behavior. The dynamic analysis result only exists when the analyzed behavior happens during the program running time. However, dynamic analysis methods cannot guarantee the code coverage and require specialized input to activate a certain part of the program. Therefore, the collected execution traces may not be the same from different executions (our method utilizes proper input to ensure the vulnerability is executed during runtime). Besides, our method relies on the foreknowledge of a vulnerability. In some productive scenarios, it is complicated to activate a vulnerability with proper input. If a vulnerability can seldom be activated during runtime, it can hardly be analyzed with our method. Also, our method is limited to the program semantics based vulnerability analysis. If a vulnerability cannot be identified with program semantics signatures, our methods may not fit. There are many types of vulnerabilities that may not be identified with such signatures [3,29], such as configuration based vulnerabilities [48], code injection vulnerabilities, and cross-site scripting vulnerabilities. For example, SQL injection [16] vulnerability is caused by failing to verify user inputs, As a result, the basic-blocks-sequence semantics comparison cannot be used to analyze SQL injection vulnerabilities. For those vulnerabilities that cannot be identified with program semantics, our method may not be applied.

7 Related Work

SSL/TLS protocols are created to protect internet communications. As an essential security infrastructure, a lot of research has been done on analyzing SSL/TLS security [2,11,19,22,35,56]. An example of SSL/TLS vulnerabilities analysis is given by Satapathy et al. [50]. The authors give a detailed analysis of the different components of SSL/TLS. On each detailed layer, both the protocol design vulnerabilities and implementation-based vulnerabilities and attacks are analyzed. Although existing work provides an evaluation basis on SSL/TLS security properties, most of them are either explanatory or prone to design based SSL/TLS vulnerabilities. Comparatively, our work is using binary semantics comparison to analyze the semantics of implementation-based SSL/TLS vulnerabilities.

Software vulnerability has been an inevitable threat to the safety of modern computer systems. The existing software vulnerability analysis methods can be divided into two categories, static analysis methods and dynamic analysis methods. Static analysis methods are capable of locating the vulnerable code by analyzing the content or structure of the program without executing it. Static analysis process usually involves manual inspection work by experienced security engineers (although many static analysis tools are claimed as "automatic", they still require users to have certain software "analysis" foreknowledge to operate the tools and verify the detection result). By employing the characteristics information of the program, such as abstract syntax tree [53], function call graph [26,52,58], and information flow [17], it is easy to locate vulnerable places or bad coding practices. For example, Pixy [25] makes use of static analysis to detect specific web application vulnerabilities. Zhang et al. [65] propose to utilize the path sensitive information to prevent web applications from being attached by remote code execution vulnerabilities. Similarly, static analysis technologies are also used to protect the low level code. Marco et al. [9] enhance the binary code safety by employing security-related flaws with static analysis methods. However, static analysis vulnerability detection suffers accuracy problems. Since all the static detection activities are relying on heuristic analysis rather than code running behavior, static analysis methods often generate inaccurate results.

Dynamic analysis methods detect software vulnerabilities based on their behavior. There are many dynamic analysis technologies, including code instrumentation, dynamic taint analysis, and so on. Code instrumentation [14,30,42] technologies dynamically insert the "analysis code" into a target program at runtime, and the target program is actually running with the inserted "analysis code". Through "analysis code", users can easily monitor or analyze the target program to finish various tasks, like program profiling, performance monitoring, vulnerability detection, and so on. Our method is making use of code instrumentation technology. Dynamic taint analysis [8,43,51] is another dynamic analysis scheme, which is used to trace runtime information flow. It marks taint source (usually user input) in a program and traces the "flow" of taint source within the program. By observing which part of computations is affected by the predefined taint sources, dynamic taint analysis is able to detect when and how the taint source triggers a possible software vulnerability. Unlike dynamic taint

analysis, we make use of the instrumentation tool to collect the vulnerability execution traces. We abstract a vulnerability execution trace as a "signature" trace and use basic-blocks-sequence semantics comparison to compare it with a target trace. With semantics comparison result, we are able to detect whether the target program contains the vulnerability or not.

Code similarity detection aims to detect whether a given code component is similar to an component in another program. It has been applied in various research scenarios. Software plagiarism detection [39,63]is an important application of code similarity technique. Jhi et al. [23,24] introduce to use value-based program characterization to identify software plagiarism. By making use of invariant critical runtime values, value-based program characterization can be used to detect pharisaism among softwares with various obfuscation methods. Similarly, third-party library detection can also be solve with code similarity schemes. LibD [27,28] is designed to identify third-party Android libraries with code similarity techniques. By making use of internal code dependencies of an Android app, LibD is able to handle the Android code whose package and method names are obfuscated. Besides, the similar research methods can be used on mobile app repackaging detection. Mobile apps can be easily repackaged by an attacker for various purposes. Zhang et at. [62] provides ViewDroid, an obfuscation-resilient method to detect mobile app repackaging. It makes use of user interfaces (views) as a new birthmark for Android apps. Through high level abstraction, ViewDroid is able to handle different code obfuscation schemes and detect repackaged apps at a large scale. Compared with ViewDroid, repackage-proofing [31] is another technique which is used on mobile app repackaging problems. Unlike ViewDroid, repackage-proofing inserts a large number of detection nodes into Android apps without introducing much overhead. Our work is related with code similarity detection and is based on the semantics of binary code. Compared with the above methods, we identify the vulnerabilities by comparing signature traces and target traces in terms of code semantics.

8 Conclusion

In this paper, we present the feasibility of analyzing implementation-based SSL/TLS vulnerabilities with basic-blocks-sequence semantics comparison. We abstract a vulnerability execution trace as a "signature". By comparing the semantics of a target program's execution trace and a vulnerability's "signature", we are able to detect whether the target program contains the vulnerability or not. We analyzed the well-known Heartbleed vulnerability and other implementation-based vulnerabilities in two popular SSL/TLS libraries, OpenSSL and MbedTLS. The evaluation result shows that our basic-blocks-sequence semantics comparison method is effective on analyzing the existence of implementation-based SSL/TLS vulnerabilities. In the future, we consider integrating basic-blocks-sequence semantics comparison with deep learning technologies to explore the possibility of increasing binary semantics comparison efficiency.

References

1. Aviram, N., et al.: {DROWN}: Breaking TLS using SSLv2. In: 25th USENIX Security Symposium (USENIX Security 16), pp. 689–706 (2016)
2. Bard, G.V.: The vulnerability of SSL to chosen plaintext attack. IACR Cryptology ePrint Archive **2004**(111) (2004)
3. Barrere, M., Badonnel, R., Festor, O.: Vulnerability assessment in autonomic networks and services: a survey. IEEE Commun. Surv. Tutorials **16**(2), 988–1004 (2013)
4. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 1–12. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055716
5. Brown, P.F., Desouza, P.V., Mercer, R.L., Pietra, V.J.D., Lai, J.C.: Class-based n-gram models of natural language. Comput. Linguist. **18**(4), 467–479 (1992)
6. Caballero, J., Poosankam, P., McCamant, S., Babi ć, D., Song, D.: Input generation via decomposition and re-stitching: finding bugs in malware. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (2010)
7. Calzavara, S., Focardi, R., Nemec, M., Rabitti, A., Squarcina, M.: Postcards from the post-http world: amplification of https vulnerabilities in the web ecosystem. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 281–298. IEEE (2019)
8. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, pp. 196–206 (2007)
9. Cova, M., Felmetsger, V., Banks, G., Vigna, G.: Static detection of vulnerabilities in x86 executables. In: 2006 22nd Annual Computer Security Applications Conference (ACSAC 2006), pp. 269–278. IEEE (2006)
10. Durumeric, Z., et al.: The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference, pp. 475–488 (2014)
11. Eldewahi, A.E., Sharfi, T.M., Mansor, A.A., Mohamed, N.A., Alwahbani, S.M.: SSL/TLS attacks: Analysis and evaluation. In: 2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE), pp. 203–208. IEEE (2015)
12. Evtyushkin, D., Riley, R., Abu-Ghazaleh, N.B., Ponomarev, D.: Branchscope: A new side-channel attack on directional branch predictor. In: Shen, X., Tuck, J., Bianchini, R., Sarkar, V. (eds.) Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018. pp. 693–707. ACM (2018)
13. Gao, D., Reiter, M.K., Song, D.: BinHunt: automatically finding semantic differences in binary programs. In: Poceedings of the 10th International Conference on Information and Communications Security (ICICS 2008) (2008)
14. Garnett, T.: Dynamic optimization if IA-32 applications under DynamoRIO. Ph.D. thesis, Massachusetts Institute of Technology (2003)
15. Gröbert, F., Willems, C., Holz, T.: Automated identification of cryptographic primitives in binary programs. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 41–60. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23644-0_3
16. Halfond, W.G., Viegas, J., Orso, A., et al.: A classification of SQL-injection attacks and countermeasures. In: Proceedings of the IEEE International Symposium on Secure Software Engineering, vol. 1, pp. 13–15. IEEE (2006)

17. Hamadou, S., Sassone, V., Palamidessi, C.: Reconciling belief and vulnerability in information flow. In: 2010 IEEE Symposium on Security and Privacy, pp. 79–92. IEEE (2010)
18. The Heartbleed Bug. https://heartbleed.com. Accessed 17 May 2023
19. Holz, R., Braun, L., Kammenhuber, N., Carle, G.: The SSL landscape: a thorough analysis of the x. 509 pki using active and passive measurements. In: Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference, pp. 427–444 (2011)
20. Secure Sockets Layer Version 3.0. https://tools.ietf.org/html/rfc6101. Accessed 17 May 2023
21. Transportation Layer Security Version 1.3. https://tools.ietf.org/html/rfc8446. Accessed 17 May 2023
22. Jarmoc, J., Unit, D.: SSL/TLS interception proxies and transitive trust. Black Hat Europe (2012)
23. Jhi, Y., Jia, X., Wang, X., Zhu, S., Liu, P., Wu, D.: Program characterization using runtime values and its application to software plagiarism detection. IEEE Trans. Software Eng. **41**(9), 925–943 (2015)
24. Jhi, Y., Wang, X., Jia, X., Zhu, S., Liu, P., Wu, D.: Value-based program characterization and its application to software plagiarism detection. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. pp. 756–765. ACM (2011)
25. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: a static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy (S&P 2006), pp. 6–pp. IEEE (2006)
26. Kinable, J., Kostakis, O.: Malware classification based on call graph clustering. J. Comput. Virol. **7**(4), 233–245 (2011)
27. Li, M., et al.: Large-scale third-party library detection in android markets. IEEE Trans. Softw. Eng. **46**(9), 981–1003 (2018)
28. Li, M., et al.: LibD: scalable and precise third-party library detection in android markets. In: Uchitel, S., Orso, A., Robillard, M.P. (eds.) Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, 20–28 May 2017, pp. 335–346. IEEE/ACM (2017)
29. Liu, B., Shi, L., Cai, Z., Li, M.: Software vulnerability discovery techniques: a survey. In: 2012 Fourth International Conference on Multimedia Information Networking and Security, pp. 152–156. IEEE (2012)
30. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices **40**(6), 190–200 (2005)
31. Luo, L., Fu, Y., Wu, D., Zhu, S., Liu, P.: Repackage-proofing android apps. In: 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28–July 1, 2016, pp. 550–561. IEEE Computer Society (2016)
32. Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S.: Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 389–400 (2014)
33. Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S.: Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. IEEE Trans. Software Eng. **43**(12), 1157–1177 (2017)

34. Mbed TLS: an implementation of the TLS and SSL protocols. https://tls.mbed.org/. Accessed 17 May 2020
35. Meyer, C., Schwenk, J.: SoK: lessons learned from SSL/TLS attacks. In: Kim, Y., Lee, H., Perrig, A. (eds.) WISA 2013. LNCS, vol. 8267, pp. 189–209. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05149-9_12
36. Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., Tews, E.: Revisiting SSL/TLS implementations: new bleichenbacher side channels and attacks. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 733–748 (2014)
37. Ming, J., Pan, M., Gao, D.: iBinHunt: binary hunting with inter-procedural control flow. In: Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC 2012) (2012)
38. Ming, J., Xu, D., Wu, D.: Malwarehunt: semantics-based malware diffing speedup by normalized basic block memoization. J. Comput. Virol. Hacking Tech. **13**(3), 167–178 (2017)
39. Ming, J., Zhang, F., Wu, D., Liu, P., Zhu, S.: Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. IEEE Trans. Reliab. **65**(4), 1647–1664 (2016)
40. Möller, B., Duong, T., Kotowicz, K.: This poodle bites: exploiting the SSL 3.0 fallback. Security Advisory (2014)
41. Nemec, M.: Challenging RSA cryptosystem implementations. Ph.D. thesis, Ca'Foscari University (2019)
42. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, 10–13 June 2007. pp. 89–100. ACM (2007)
43. Newsome, J., Song, D.X.: Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA. The Internet Society (2005)
44. National Vulnerability Database. https://nvd.nist.gov/. Accessed 17 May 2023
45. Oehlert, P.: Violating assumptions with fuzzing. IEEE Secur. Priv. **3**(2), 58–62 (2005)
46. OpenSSL: a toolkit for the transport layer security (TLS) and secure sockets layer (SSL) protocols. https://www.openssl.org/. Accessed 17 May 2023
47. Poskanzer, J.: mini httpd: small http server. https://acme.com/software/mini_httpd/
48. Ramakrishnan, C., Sekar, R.: Model-based analysis of configuration vulnerabilities 1. J. Comput. Secur. **10**(1–2), 189–209 (2002)
49. Ronen, E., Gillham, R., Genkin, D., Shamir, A., Wong, D., Yarom, Y.: The 9 lives of bleichenbacher's cat: new cache attacks on TLS implementations. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 435–452. IEEE (2019)
50. Satapathy, A., LM, J.L.: A comprehensive survey on SSL/TLS and their vulnerabilities. Int. J. Comput. Appl. **153**(5), 31–38 (2016)
51. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposiums on Security and Privacy, pp. 317–331. IEEE (2010)
52. Shang, S., Zheng, N., Xu, J., Xu, M., Zhang, H.: Detecting malware variants via function-call graph similarity. In: 2010 5th International Conference on Malicious and Unwanted Software, pp. 113–120. IEEE (2010)

53. Sotirov, A.I.: Automatic vulnerability detection using static source code analysis. Ph.D. thesis, Citeseer (2005)
54. Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Pearson Education, London (2007)
55. Transportation Layer Security. https://en.wikipedia.org/wiki/Transport_Layer_Security. Accessed 27 May 2023
56. Wagner, D., Schneier, B., et al.: Analysis of the SSL 3.0 protocol. In: The Second USENIX Workshop on Electronic Commerce Proceedings, vol. 1, pp. 29–40 (1996)
57. Wang, L., Xu, D., Ming, J., Fu, Y., Wu, D.: Metahunt: towards taming malware mutation via studying the evolution of metamorphic virus. In: Proceedings of the 3rd ACM Workshop on Software Protection, pp. 15–26 (2019)
58. Xu, M., et al.: A similarity metric method of obfuscated malware using function-call graph. J. Comput. Virol. Hacking Tech. **9**(1), 35–47 (2013)
59. Xu, Z., Chen, B., Chandramohan, M., Liu, Y., Song, F.: Spain: security patch analysis for binaries towards understanding the pain and pills. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 462–472. IEEE (2017)
60. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, pp. 590–604 (2014)
61. Yamaguchi, F., Lottmann, M., Rieck, K.: Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 359–368 (2012)
62. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: Viewdroid: towards obfuscation-resilient mobile application repackaging detection. In: Ács, G., Martin, A.P., Martinovic, I., Castelluccia, C., Traynor, P. (eds.) 7th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec 2014, Oxford, United Kingdom, 23–25 July 2014, pp. 25–36. ACM (2014)
63. Zhang, F., Wu, D., Liu, P., Zhu, S.: Program logic based software plagiarism detection. In: 25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, 3–6 November 2014, pp. 66–77. IEEE Computer Society (2014)
64. Zhang, T., Zhang, Y., Lee, R.B.: CloudRadar: a real-time side-channel attack detection system in clouds. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 118–140. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_6
65. Zheng, Y., Zhang, X.: Path sensitive static analysis of web applications for remote code execution vulnerability detection. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 652–661. IEEE (2013)