Research Article

CAFA: A Checksum-Aware Fuzzing Assistant Tool for Coverage Improvement

Xiaolong Liu , **Qiang Wei** , **Qingxian Wang**, **Zheng Zhao** , and **Zhongxu Yin**

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450002, China

Correspondence should be addressed to Qiang Wei; xdweiqiang@163.com

Received 30 May 2018; Accepted 4 October 2018; Published 16 October 2018

Academic Editor: Emanuele Maiorana

Copyright © 2018 Xiaolong Liu et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Fuzzing is an effective technique to discover vulnerabilities that involves testing applications by constructing invalid input data. However, for applications with checksum mechanism, fuzzing can only achieve low coverage because samples generated by the fuzzer are possibly incapable of passing the checksum verification. To solve this problem, most current fuzzers advise the user to comment out the checksum verification code manually, but it requires considerable time to audit the source code to identify the checksum point corresponding to checksum verification. In this paper, we present a novel approach based on taint analysis to identify the checksum point automatically. To implement this approach, the checksum-aware fuzzing assistant tool (CAFA) is designed. After the checksum point is identified, the application is statically patched in an antilogical manner at the checksum point. The fuzzing tool then tests the patched program to bypass the checksum verification. To evaluate CAFA, we use it to assist the American Fuzzy Lop (AFL) tool in fuzzing eight real-world applications with known input specification. The experimental results show that CAFA can accurately and quickly identify the checksum points and greatly improve the coverage of AFL. With the help of CAFA, multiple buffer overflow vulnerabilities have been discovered in the newest ImageMagick and RAR applications.

1. Introduction

Fuzz testing proposed by Professor Barton Miller in the 1990s [1] was used in the simple form of pure black-box fuzzing to investigate the reliability of UNIX applications. To solve the blindness problem of the original fuzzing, white-box fuzzing (such as SAGE [2], BAP [3], and KLEE [4]) based on symbolic execution [5] was then proposed. White-box fuzzing presented the input as symbols and explored different paths by solving path constraints, so that it greatly improved the coverage. However, white-box fuzzing requires considerable time for heavyweight application analysis and constraint solving, so it cannot scale to large, real-world applications [6]. Grey-box fuzzing lies somewhere in between, not requiring heavyweight application analysis. A representative example of a grey-box fuzzing tool is American Fuzzy Lop (AFL) [7] that has discovered hundreds of vulnerabilities in widely used applications. AFL uses lightweight instrumentation to obtain coverage information and a genetic algorithm to determine which sample to fuzz next.

However, all three fuzzing approaches above achieve low coverage for applications with checksum mechanism. This mechanism is used to protect the integrity of input samples: it checks whether a sample has been damaged by comparing checksum values. When a sample is detected as being damaged, the application will report an error and exit. Since most samples mutated by a fuzzer have corrupted checksum values, they will fail to pass the checksum verification. As a result, the fuzzer fails to test deeper paths, leading to low coverage.

There are three main approaches for solving the checksum problem. (1) The first approach (e.g., [8, 9]) is to generate and solve the constraints of the checksum verification through symbolic execution engines and constraint solvers. (2) The second approach (e.g., [10, 11]) is to construct a grammar that delineates the format information of the input values. This type of fuzzing tool can generate each well-formed sample based on the grammar. (3) The third approach recommends that users manually patch the program at the checksum point corresponding to the checksum verification, such as AFL and libFuzzer [12]. When a crashed sample is

found, the checksum field of the crashed sample needs to be fixed in order to reduce false positives.

Because the application of symbolic techniques to large real-world programs remains a challenge [5] and the constraints derived from some checksum algorithms is too complex for modern constraint solvers [8], the first approach is of limited use. The high time consumption to manually construct a full grammar for the input also limits the application of the second approach. Techniques such as symbolic execution [13] and deep learning [14] have been attempted to extract the input grammar automatically from samples, but these techniques are time-consuming and inaccurate. Even if we know the input file specification and can construct an accurate input grammar, the second approach is slower than the third approach because the second approach needs to calculate the correct checksum value for each sample based on the grammar, while the third approach only repairs the checksum fields for the crashed samples. The third approach is the fastest at runtime, but it requires a considerable amount of time to manually audit the source code to identify the checksum point beforehand and is not yet applicable to closed-source applications.

In this paper, we attempt to solve the problem of the third approach. Our goal is to propose a method to increase the automation of identifying the checksum point, and this method works for both open-source and closed-source applications. We propose a novel approach based on taint analysis and design a checksum-aware fuzzing assistant tool (CAFA, pronounced “kah-fah”) to automatically identify the checksum point for the application with known input specification. According to the input specification, the malformed sample can be constructed accurately from the well-formed sample by breaking the checksum relationship. The well-formed sample and the malformed sample are input into the application for two dynamic tests. During the dynamic test, each instruction is analyzed by dynamic instrumentation. Based on the identification algorithm and the collected dynamic information, the checksum point can be automatically identified. The application is statically patched in an antilogical manner at the checksum point. The fuzzing tool tests this patched application, so the checksum verification can be bypassed. However, the crashed samples found during the test need to be repaired according to the input specifications.

This paper makes the following specific contributions:

- (1) We propose two strategies for identifying checksum points depending on the checksum verification algorithm used in the application. The CRC32-S strategy is suitable for applications using the standard CRC32 checksum verification algorithm. This strategy can be simply configured, and it can rapidly identify the checksum point. The Taint-S strategy is appropriate for applications using a general checksum verification algorithm, so it is more broadly applicable.
- (2) We design a checksum-aware fuzzing assistant tool (CAFA) that can automatically identify the checksum point in an application. With the help of CAFA, the coverage of AFL for the application with the checksum mechanism has improved significantly. As coverage increases, more vulnerabilities are discovered.
- (3) To foster further research in this area and in support of open science, we are open-sourcing our CAFA prototype, which is available at <https://github.com/CAFA1/CAFA.git>. Some test cases are also provided for verification.

TABLE 1: PNG chunk specification.

Name	Size	Description
Length	4 bytes	Length of data field
Type	4 bytes	Chunk type code
Data	n bytes	Data bytes
CRC	4 bytes	CRC of type and data

2. Our Proposed Approach

2.1. Problem Scope. In some network protocols, the checksum mechanism is used to protect the integrity of input data. For example, each network packet in the TCP protocol [15] has a checksum field. The checksum is recalculated at the receiver and compared to the received checksum field. If the two values are the same, the packet is well-formed, otherwise the packet is discarded. The checksum mechanism is also used in some image file specifications (e.g., PNG [16], MNG [17]) and some compressed file specifications (e.g., GZIP [18], RAR [19], and ZIP [20]). The common checksum verification algorithms used in practice include CRC32, TCP/IP checksum, fletcher32, Adler32, and tar checksum.

We take the real-world application ImageMagick parsing the PNG specification as an example to illustrate the checksum verification process. A PNG file contains a preceding 8-byte PNG signature followed by several data chunks, including IDHR, IDAT, IEND, and other types of chunks. Each chunk includes a length field, a type field, a data field, and a CRC check field (see Table 1). According to the PNG specification, the value stored in the CRC field (checksum field) is calculated from the type field and the data field via the standard CRC32 algorithm. When the ImageMagick application parses a PNG file, it recalculates the CRC value of each data chunk and compares the value with the corresponding CRC field at the checksum point. If the two values are inconsistent, the input sample will fail to pass the checksum verification and cause the application to report an error and exit (see Figure 1).

Due to the blindness of random mutation, a fuzzer may tamper with any bytes in the type, data, or CRC fields of a chunk and thus is highly likely to corrupt the calculated checksum. Most samples cannot pass the checksum verification, so deep code cannot be tested. To bypass the checksum verification, AFL advises users to comment out the source code at each checksum point in the libpng-nocrc.patch. However, this approach requires an extensive manual analysis of the libpng source code and is only valid for a particular version of libpng.

```

pngutil.c:167
167  if (png_crc_error(png_ptr))//checksum point
168  {
169      if (((png_ptr->chunk_name[0] & 0x20) &&
170          !(png_ptr->flags & PNG_FLAG_CRC_ANCILLARY_NOWARN)) ||
171          (!(png_ptr->chunk_name[0] & 0x20) &&
172          (png_ptr->flags & PNG_FLAG_CRC_CRITICAL_USE)))
173      {
174          png_chunk_warning(png_ptr, "CRC error");
175      }
176      else
177      {
178          png_chunk_error(png_ptr, "CRC error");
179      }
180      return (1);
181  }

```

The corresponding assembly code in libpng12.so.0.46.0:

```

.text:00007966      mov     [esp+2Ch+var_2C], esi
.text:00007969      call   png_crc_error
.text:0000796E      xor     edx, edx
.text:00007970      test    eax, eax
.text:00007972      jz      short loc_799D ; checksum point
.text:00007974      test    byte ptr [esi+178h], 20h
.text:0000797B      jnz     short loc_79A8
.text:0000797D      test    byte ptr [esi+0C9h], 4
.text:00007984      jnz     short loc_79B1

```

FIGURE 1: The checksum point in libpng12.so.0.46.0.

In this paper, we focus on applications that parse known file specifications and use the checksum mechanism. We design a checksum-aware fuzzing assistant tool (CAFA) to identify checksum points automatically and assist a fuzzer in bypassing the checksum verification in these applications. For example, CAFA can identify the checksum point for the ImageMagick application.

2.2. The Workflow of CAFA. In general, the workflow of CAFA is divided into four stages (see Figure 2): identifying checksum points, patching, fuzzing, and repairing crashed

samples. We also take the ImageMagick application parsing the PNG specification as an example to illustrate the workflow of CAFA.

2.2.1. Identifying Checksum Points. We accurately construct a malformed sample by tampering with any bytes of the data protected by the checksum field or the checksum field from the well-formed sample based on the PNG file specification. Then the ImageMagick application run the well-formed sample and the malformed sample respectively under dynamic instrumentation monitoring. We propose two

FIGURE 2: Workflow of CAFA.

FIGURE 3: Antilogical patching.

strategies for identifying checksum points based on taint analysis (see Section 3 for details). The checksum point is a special branch point (e.g., JZ, JNZ) where the well-formed sample follows the same branch and the malformed sample follows the other branch (see Figure 3). The experiment shows that the checksum point is located in the libpng.so library, and the corresponding conditional jump instruction is “0xb7747972: je 0xb774799d”. Because libpng library is open source, we can further locate the corresponding line of the source code, that is, “if (png_crc_error(png_ptr))” in line 167 of the pngutil.c file (see Figure 1).

2.2.2. Patching. At the checksum point, the libpng library is statically patched in an antilogical manner. For example, we change the “je 0xb774799d” instruction to a “jne 0xb774799d” instruction in the binary. In the original application, the well-formed sample can pass the checksum verification, whereas the malformed sample fails to pass (see Figure 3(a)). However, after patching, the well-formed sample fails to pass the checksum verification, whereas the malformed sample can pass (see Figure 3(b)). The logic of the judgment is reversed before and after patching, so we call it an antilogical patch. In addition, if the fuzzing tool (like libFuzzer) needs to patch the source code, we can change the

source line “if (png_crc_error(png_ptr))” to “if (!png_crc_error(png_ptr))”.

2.2.3. Fuzzing. The original libpng.so library is replaced with the patched one, and then the ImageMagick application is tested by a fuzzing tool (such as AFL). Most PNG samples generated by the fuzzing tool would have great difficulty passing the checksum verification in the original application. However, after the libpng.so library has been anti-logically patched, most of the mutated PNG samples can pass the checksum verification to test deeper code. As a result, the coverage of the fuzzing tool is improved. In theory, the greater the code coverage, the greater the chance of finding vulnerabilities.

2.2.4. Repairing Crashed Samples. The crashed sample actually crashes the patched program, not the original program. If the crashed sample fails to pass the checksum verification and cannot cause the original program to crash, it will cause a false positive. Therefore, the checksum field of the crashed sample needs to be repaired to allow it to pass the checksum verification in the original application. The checksum field can be recalculated according to the CRC32 algorithm defined in the PNG specification. If a repaired

Name	Value	Start	Size	Unit
struct PNG_SIGNATURE sig		0h	8h	Fg
struct PNG_CHUNK chunk[0]	IHDR (Critical, Public, ...)	8h	19h	Fg
uint32 length	13	8h	4h	Fg
union CTYpe type	IHDR	Ch	4h	Fg
struct PNG_CHUNK_IHDR ihdr	100 x 100 (x8)	10h	Dh	Fg
uint32 crc	5589CA88h	1Dh	4h	Fg
struct PNG_CHUNK chunk[1]	IDAT (Critical, Public, ...)	21h	1AD2h	Fg
struct PNG_CHUNK chunk[2]	IEND (Critical, Public, ...)	1AF3h	Ch	Fg

FIGURE 4: Locating the checksum field in a PNG file.

crashed sample can crash the original application, we will report the vulnerability triggered by these crashed sample.

3. Identification Strategies

In this section, we describe the identification strategies implemented in CAFA. Depending on the checksum verification algorithm used in the application, we propose two identification strategies: the CRC32-S strategy for applications using the standard CRC32 checksum verification algorithm and the Taint-S strategy for applications using a general checksum verification algorithm. Before introducing these identification strategies, we formalize the checksum process.

3.1. Checksum Verification Process. We assume that a file with a checksum mechanism consists of three components: raw data related to the checksum calculation (denoted by D), a checksum field (denoted by C) and other data (denoted by O). For example, a PNG file can be represented by $O(DC)^*$ in a regular expression. The application recalculates the checksum value from D and compares the recalculated value with the checksum field C stored in the input file. Let R denote the recalculated checksum value determined based on D. The relationship between R and D is expressed as $R = f_d(D)$, where f_d denotes the function implemented in the checksum verification algorithm. For example, for the PNG specification, f_d denotes the crc32 function. Let C' represent the converted value of C before the checksum verification. The relationship between C' and C is expressed as $C' = f_c(C)$, where f_c denotes the function that converts the checksum field stored in the input file into an integer in the memory. For example, f_c is an inverse function for the PNG specification because the checksum field is stored in big endian format in a PNG file, but it should be reversed to the little endian format in the memory before the comparison. In Figure 4, the real checksum value calculated from the data of the PNG chunk[0] is 0xbb20485f (little endian), and the value stored in the PNG file is "0xbb,0x20,0x48,0x5f" (big endian) in hexadecimal format.

At a checksum point, the application compares the recalculated checksum value with the checksum field of the file. If they do not match, the sample is malformed. The check

TABLE 2: Intermediate language CAFA-IL.

application	::= ins*
ins	::= cji cli oti
cji	::= conditional jump instructions
cli	::= constant instructions
oti	::= other instructions
operand p	::= register memory
register r	::= eax ebx ecx edx eflags ...
taint status	::= T F
taint labels	::= labels of the taint source
τ_Δ	::= map of register or memory addresses to taint statuses
$\mathcal{F}_\Delta$	::= map of register or memory addresses to taint labels
$\diamond_b$	::= typical OR operator
$\diamond_x$	::= typical XOR operator
$\diamond_u$	::= typical UNION operator for sets

condition can be formally expressed as $P: R == C'$, which can be extended to $P: f_d(D) == f_c(C)$. We find that an application will always include a conditional jump instruction to perform the check condition P, such as a jz or jnz instruction. From the check condition P, we can infer the following heuristics for identifying the checksum point:

- H1.** A conditional jump instruction is present, and whether it is taken is affected by the sample.
- H2.** Whether the conditional jump instruction is taken is related to a certain number of bytes in the sample.
- H3.** When a well-formed sample is run, the conditional jump instruction exhibits the opposite behavior from that observed when a malformed sample is run.

3.2. Dynamic Taint Analysis. The heuristics introduced above are implemented based on taint analysis. To illustrate the taint analysis policy for identifying checksum points, we design the intermediate language CAFA-IL, as presented in Table 2. An application consists of a sequence of instructions, including conditional jump instructions, constant instructions, and

TABLE 3: Operational semantics of CAFA-IL.

$\tau'_\Delta = \tau_\Delta, \mathcal{F}'_\Delta = \mathcal{F}_\Delta$	
$\langle \tau_\Delta, \mathcal{F}_\Delta \rangle \rightsquigarrow \langle \tau'_\Delta, \mathcal{F}'_\Delta \rangle$	(cji)
$\tau'_\Delta(p) = F, \mathcal{F}'_\Delta(p) = \emptyset$	
$\langle \tau_\Delta, \mathcal{F}_\Delta \rangle \rightsquigarrow \langle \tau'_\Delta, \mathcal{F}'_\Delta \rangle$	(cli)
$\tau'_\Delta(p_{dst1}) = \Diamond_b(\tau_\Delta(p_{src1}), \dots), \mathcal{F}'_\Delta(p_{dst1}) = \Diamond_u(\mathcal{F}_\Delta(p_{src1}), \dots)$	
$\langle \tau_\Delta, \mathcal{F}_\Delta \rangle \rightsquigarrow \langle \tau'_\Delta, \mathcal{F}'_\Delta \rangle$	(oti)

other instructions. Constant instructions always produce constant results regardless of the operands, such as xor eax, eax and sub eax, eax. The taint state is described by two parameters: the current taint status map (τ_Δ) and the current taint label map ($\mathcal{F}_\Delta$) of the memory and registers. For example, if the register eax is tainted by the second byte and the third byte in the file, this situation can be expressed as $\tau_\Delta(\text{eax}) = T, \mathcal{F}_\Delta(\text{eax}) = (2, 3)$.

The operational semantics for the three types of instructions are shown in Table 3. Each rule statement has the following form:

$$\frac{\text{propagation rule}}{\langle \text{current state} \rangle \rightsquigarrow \langle \text{end state} \rangle} \quad (1)$$

Each instruction may change the current taint state based on its propagation rule. The purpose of the dynamic taint analysis is to track the flow of the taint source when executing the instructions.

Unlike Dytan [21], we do not consider tainting due to the control flow to reduce overtainting. Therefore, the propagation rule of the operational semantic for the conditional jump instruction (**cji**) is that the taint state ($\langle \tau_\Delta, \mathcal{F}_\Delta \rangle$) remains unchanged. For example, if the eflags register of the “je 0xb774799d” instruction is tainted, the taint status of the eflags source operand will not propagate to the eip destination operand containing the address of the next instruction to be executed.

The propagation rule of the operational semantic for the constant instruction (**cli**) is that the taint status $\tau'_\Delta(p)$ and taint labels $\mathcal{F}'_\Delta(p)$ of the destination operand p are cleared because the destination operand is set to a constant value regardless of whether the source operands are tainted. For example, if the eax register of the “xor eax, eax” instruction is tainted, the taint status of the eax register operand will be set to false ($\tau'_\Delta(\text{eax}) = F$) and the taint labels of the eax register operand will be set to empty ($\mathcal{F}'_\Delta(\text{eax}) = \emptyset$) after this instruction is executed.

The propagation rule of the operational semantic for other instructions (**oti**) is that the taint status ($\tau'_\Delta(p_{dsti})$) of each destination operand (p_{dsti}) is a Boolean value resulting from performing the OR operation ($\Diamond_b$) on the taint status ($\tau_\Delta(p_{srcj})$) of each source operator (p_{srcj}), and the taint labels ($\mathcal{F}'_\Delta(p_{dsti})$) of each destination operand (p_{dsti}) are a set resulting from performing the UNION operation ($\Diamond_u$) on the set of taint labels ($\mathcal{F}_\Delta(p_{srcj})$) for each source operand (p_{srcj}). We use the “add eax, ebx” instruction as an example. The eax source operand is tainted by the first byte and the second byte in the input sample ($\tau_\Delta(\text{eax}) = T, \mathcal{F}_\Delta(\text{eax}) =$

$(1, 2)$), and the ebx source operand is tainted by the third byte and the fourth byte in the input sample ($\tau_\Delta(\text{ebx}) = T, \mathcal{F}_\Delta(\text{ebx}) = (3, 4)$). After the instruction is executed, the taint status of the eax destination operand is $\tau'_\Delta(\text{eax}) = \Diamond_b(\tau_\Delta(\text{eax}), \tau_\Delta(\text{ebx})) = T$, and the taint labels are $\mathcal{F}'_\Delta(\text{eax}) = \Diamond_u(\mathcal{F}_\Delta(\text{eax}), \mathcal{F}_\Delta(\text{ebx})) = (1, 2, 3, 4)$. In addition, there is an implied eflags destination operand whose taint status and taint labels will be $\tau'_\Delta(\text{eflags}) = \Diamond_b(\tau_\Delta(\text{eax}), \tau_\Delta(\text{ebx})) = T$ and $\mathcal{F}'_\Delta(\text{eflags}) = \Diamond_u(\mathcal{F}_\Delta(\text{eax}), \mathcal{F}_\Delta(\text{ebx})) = (1, 2, 3, 4)$ after the instruction is executed.

Let CP denote the set of checksum points and $\Pi(s, i)$ denote whether conditional jump instruction i is taken when sample s is input. Then, we can formally restate the heuristics as follows:

H1. $\text{CP} \subseteq \{\text{cji} \mid \tau_\Delta(\text{eflags}) == T\}$.

H2. $\text{CP} \subseteq \{\text{cji} \mid |\mathcal{F}_\Delta(\text{eflags})| > \theta\}$, where θ denotes a constant threshold.

H3. $\text{CP} \subseteq \{\text{cji} \mid \Diamond_x(\Pi(w, \text{cji}), \Pi(m, \text{cji})) == 1\}$, where w denotes a well-formed sample and m denotes a malformed sample.

3.3. CRC32-S Strategy. For the application that uses the standard CRC32 algorithm as the checksum verification algorithm, we propose the CRC32-S strategy to identify the checksum point. If the checksum verification is implemented by the function of crc32 in the libz.so library, we can directly introduce the taint source from this function.

Since the recalculated checksum value is the return value of the crc32 function ($R = \text{crc32}(D)$), the return value can be introduced as the taint source with a single taint label. The eflags register of the conditional jump instruction at the checksum point must be tainted by recalculated checksum value, so rule H1 can be used. However, because there is only one taint label, rule H2 does not apply. Furthermore, if the conditional jump instruction at a checksum point is taken (not taken) when a well-formed sample is parsed, it must be not taken (taken) when a malformed sample is parsed. Thus, as shown in Algorithm 1, rules H1 and H3 are used to identify the checksum point. Finally, the checksum point is identified from the tainted conditional jump instructions that behave differently between the two executions.

3.4. Taint-S Strategy. For the application that uses a general checksum algorithm, we propose the Taint-S strategy to identify the checksum point. The Taint-S strategy applies as long as the checksum algorithm (f_d) satisfies the following check condition: $P: f_d(D) == f_c(C)$. As far as we know, all checksum verifications satisfy this check condition P.

Input: a well-formed sample (w)
and a malformed sample (m)
Output: CP (checksum points)
(1) $SET1 \leftarrow \emptyset, SET2 \leftarrow \emptyset, CP \leftarrow \emptyset$
(2) Run the well-formed sample (w) for taint analysis
(3) $SET1 \leftarrow \{c_{ji} \mid \tau_{\Delta}(eflags) == T\}$
(4) Run the malformed sample (m) for taint analysis
(5) $SET2 \leftarrow \{c_{ji} \mid \tau_{\Delta}(eflags) == T\}$
(6) $CP \leftarrow \{c_{ji} \mid \Diamond_x(\Pi(w, c_{ji}), \Pi(m, c_{ji})) == 1\}$

ALGORITHM 1: Identifying checksum points using the CRC32-S strategy.

Input: a well-formed sample (w), a malformed sample (m)
and the configured taint source range (offset, length)
Output: CP (checksum points)
(1) $SET1 \leftarrow \emptyset, SET2 \leftarrow \emptyset, SET3 \leftarrow \emptyset, SET4 \leftarrow \emptyset, \theta = \text{length} * 0.5$
(2) Run the well-formed sample (w) for taint analysis
(3) $SET1 \leftarrow \{c_{ji} \mid \tau_{\Delta}(eflags) == T\}$
(4) Run the malformed sample (m) for taint analysis
(5) $SET2 \leftarrow \{c_{ji} \mid \tau_{\Delta}(eflags) == T\}$
(6) $SET3 \leftarrow \{c_{ji} \mid |\mathcal{F}_{\Delta}(eflags)| > \theta\}$
(7) $CP \leftarrow \{c_{ji} \mid \Diamond_x(\Pi(w, c_{ji}), \Pi(m, c_{ji})) == 1 \text{ and } c_{ji} \in SET3\}$

ALGORITHM 2: Identifying checksum points using the Taint-S strategy.

Under this strategy, the policy for taint source introduction is to introduce data (D) protected by the checksum field or the checksum field (C) as the taint source. The location of C or D can be found based on the part of checksum verification in the file specification. If the data (D) protected by the checksum field is specified as the taint source, the recalculated checksum value ($f_d(D)$) is tainted by the taint source. If the checksum field (C) is specified as the taint source, the converted checksum value ($f_c(C)$) is tainted by the taint source. Because one side of the check condition (P) equation is always tainted in both cases, the $eflags$ register of the conditional jump instruction is tainted. Therefore, rule H1 applies. Both the recalculated checksum value ($f_d(D)$) and the converted checksum value ($f_c(C)$) are calculated from multiple bytes, so whether the conditional jump instruction is taken is related to a certain number of bytes in the sample. For the instruction whose taint labels exceeds a certain threshold, we call it the highly tainted instruction. While running the malformed sample, the highly tainted conditional jump instructions are collected. Therefore, rule H2 applies. Usually, the threshold in H2 is set to 50% of the length of the taint source, but it can also be set by the user. When the well-formed sample is run, the check condition (P) is satisfied. Since the malformed sample is constructed by tampering with D or C , the check condition (P) cannot be satisfied when running the malformed sample. Therefore, rule H3 applies.

As shown in Algorithm 2, rules H1, H2, and H3 are used to identify the checksum points. Finally, the checksum point is identified from the tainted conditional jump instructions that behave differently between the two executions and have a certain number of taint labels exceeding the threshold.

4. Implementation

4.1. Malformed Sample Construction. Based on the public file specification or appropriate tools, we can locate the checksum field C or data D protected by the checksum field. Then a malformed sample that fails to pass the checksum verification can be constructed accurately from a well-formed sample by tampering with any bytes of C or D . For example, we can use the 010 editor tool [22] with the PNG template to locate the checksum field of a PNG file (see Figure 4), and use the Wireshark tool [23] to locate the checksum field of a TCP packet.

4.2. Dynamic Binary Instrumentation. The execution monitor of CAFA is implemented as a Pin plugin (Pintool) [24]. Pin is a popular dynamic binary instrumentation (DBI) tool that provides a wide range of instrumentation granularity. Pin offers an instruction instrumentation mode that allows CAFA to perform an operational semantic (shown in Table 3) at each instruction. The image instrumentation mode allows CAFA to inspect and instrument an entire image. This mode is used to catch the `libz.so` module in which the `crc32` function is located. The routine instrumentation mode allows CAFA to inspect and instrument an entire routine. When the `libz.so` module is first loaded, this mode is used to instrument the `crc32` function, and the analysis code introduces the return value of the `crc32` function as the taint source under the CRC32-S strategy. Under the Taint-S strategy, CAFA instruments the system call to register two notification functions that are called immediately before and after the execution of the system call. When the “read” system call is caught,

the notification function introduces the taint source into the destination memory and marks the destination memory with the taint labels based on the offset in the file.

Drawing on part of the BAP [3] code, we have written approximately 6000 lines of C++ code to implement our Pintool to perform dynamic taint analysis. After a sample has been run, the Pintool produces two log files. One is a trace file logging the tainted conditional jump instructions for which the eflags register is tainted (H1) and their behavior (H3). The other file logs the highly tainted conditional jump instructions (H2). We have also written 300+ lines of Python script to schedule the Pintool and select the CRC32-S strategy or Taint-S strategy to analyze the two log files.

4.3. Patching and Fuzzing. After identifying the checksum points, we adopt a patching script to patch the binary in an anti-logical manner. For example, the patching script changes a “JE/JZ” instruction to a “JNE/JNZ” instruction or vice versa. If the application is open source, the corresponding source line can be located based on the symbol information. After patching the source file at this source line, the patched application can be obtained by recompiling the patched source file. Then the patched application is tested with a fuzzer such as AFL. For the closed-source application, the QEMU “user emulation” mode of AFL can be used to test the patched application. For the open-source application, we need to recompile the application with the afl-gcc or afl-g++ compiler to obtain the patched application and then test it with AFL.

4.4. Repairing Crashed Samples. For the crashed samples found during the fuzzing test, we can take full advantage of the checksum algorithm specified in the input specification to repair their checksum fields. For example, the checksum algorithm for PNG files is CRC32, so a python script that implements the standard CRC32 algorithm can repair the checksum field. Because CAFA relies on the public input specification rather than the application, it can repair the crash samples regardless of whether the program is open-source or closed-source. For example, CAFA can repair the crashed samples that cause the open-source application “pngcheck” to crash based on the CRC32 algorithm defined in the PNG file specification, and can also repair the crashed samples that cause the closed-source application “rar” to crash based on the CRC32 algorithm defined in the rar file specification.

5. Evaluation

In this section, we report experiments performed on eight real-world applications to evaluate CAFA. In Section 5.1, the experimental hardware environment is introduced. In Section 5.2, the time cost and accuracy of checksum point identification are evaluated. In Section 5.3, the effect of CAFA on the coverage is evaluated. In Section 5.4, vulnerabilities discovered under the aid of CAFA are presented.

5.1. Experimental Setup. For comparison with the TaintScope [8], we selected a similar experimental environment using a

TABLE 4: Applications tested in our experiments.

Input specification	Application	Version
PNG	ImageMagick	7.0.7-2 (libpng 1.2.46)
PNG	optipng	0.7.6 (libpng 1.2.46)
PNG	pngcheck	2.3.0 (libpng 1.2.46)
GZIP	gzip	1.4
ZIP	unzip	6.00
TAR	tar	1.26
RAR	rar	1.26
UDP/TCP/IP/IGMP	tcpdump	4.9.0

machine with 4 GB of memory and an Intel Core 2 Duo CPU @2.4 GHz running Ubuntu 12.04 (32-bit). The applications listed in Table 4 were chosen as the target applications because they use the checksum mechanism and their input specifications are public.

5.2. Checksum Point Identification. Next, we evaluate the time costs and accuracy of identifying checksum points. Our Pintool, tested applications, well-formed samples and malformed samples have all been submitted to the GitHub repository.

5.2.1. Time Cost. Under the CRC32-S strategy, we need to configure the instrumented library and checksum function because the return value of the checksum function is introduced as the taint source. But we need to configure the taint source range as the checksum field (C) under the Taint-S(C) strategy or as the data (D) protected by the checksum field under the Taint-S(D) strategy. The first three applications in Table 5 (ImageMagick, optipng, and pngcheck) use the crc32 function of the libz.so library to perform the checksum verification, so the CRC32-S strategy can be applied to them.

For each application, the tested samples are the same for the Taint-S and CRC32-S strategies, so the time costs of the three strategies are comparable. As shown in Table 5, the CRC32-S strategy is the fastest, and the Taint-S(C) strategy is faster than the Taint-S(D) strategy. During the dynamic analysis process, each taint propagation instruction requires additional multiple analysis instructions to perform the taint propagation, so the test time is largely dependent on the number of taint propagation instructions. In addition, the execution time of different applications is also related to the number of execution instructions.

For example, when testing the optipng application, the Taint-S(D) strategy is approximately 16 times the test time for the Taint-S(C) strategy. The number of execution instructions for the two strategies is the same, but the number of taint propagation instructions for the Taint-S(D) strategy (64443746) is about 14 times that of the Taint-S(C) strategy (450169). Intuitively, the wider the taint source range, the more the number of taint propagation instructions. The taint source under the Taint-S(D) strategy contains 22 taint labels, but the taint source under the Taint-S(C) strategy contains only 4 taint labels. Under the CRC32-S strategy, there is only one taint label and 469328 taint propagation instructions.

TABLE 5: Time costs of the identification strategies.

application/input specification	strategy	Taint labels	Execution instructions	Taint propagation instructions	Time(s)
ImageMagick/PNG	Taint-S(D)	22	30208236	257593	39.6
	Taint-S(C)	4	30208236	246218	39.064
	CRC32-S	1	26930357	171682	34.852
optipng/PNG	Taint-S(D)	22	9757991	6443746	420.204
	Taint-S(C)	4	9757991	450169	15.12
	CRC32-S	1	9481512	469328	13.208
pngcheck/PNG	Taint-S(D)	22	93055	12557	4.94
	Taint-S(C)	4	93055	3600	4.696
	CRC32-S	1	66818	7579	3.884
tcpdump/UDP	Taint-S(D)	8	2705841	119554	17.272
	Taint-S(C)	2	2706099	118972	17.14
tcpdump/TCP	Taint-S(D)	20	2683483	123458	17.756
	Taint-S(C)	2	2677279	118766	17.512
tcpdump/IP	Taint-S(D)	20	2675810	138548	18.508
	Taint-S(C)	2	2675552	119072	17.688
tcpdump/IGMP	Taint-S(D)	16	2718334	130481	17.94
	Taint-S(C)	2	2716120	120090	17.432
tar/TAR	Taint-S(D)	500	372314	28730	16.368
	Taint-S(C)	8	372314	20786	9.872
gzip/GZIP	Taint-S(D)	33	308253	8756	4.452
	Taint-S(C)	4	308253	8387	4.356
unzip/ZIP	Taint-S(D)	38	287077	7643	9.552
	Taint-S(C)	4	287077	4631	9.228
rar/RAR	Taint-S(D)	52	291489	75422	12.408
	Taint-S(C)	2	291489	22884	10.128

Although the number of taint propagation instructions is a little greater under the CRC32-S strategy than under the Taint-S(C) strategy, the CRC32-S strategy is a little faster, because the CRC32-S strategy executes fewer instructions and applies only rules H1 and H3 to identify the checksum points.

In addition, the time costs of lines 4, 9, 10, and 11 are much lower than those of the other lines, because there are far fewer execution instructions and taint propagation instructions in the pngcheck, tar, gzip, and unzip applications than in other applications. For example, when testing the pngcheck application under the Taint-S(C) strategy, there are only 93055 execution instructions and 3600 taint propagation instructions.

The average time cost for identifying checksum points using TaintScope is several minutes, markedly longer than that of CAFA. For example, when testing the ImageMagick application, CAFA is approximately three times faster than TaintScope. Indeed, TaintScope tested six known file specifications, but it did not use the specification. To collect the always-taken and always-not-taken conditional jump instructions, TaintScope needed to run the program many times to parse some well-formed samples and more than 10 malformed samples. CAFA takes full advantage of the known file specification and therefore can construct only one malformed sample by tampering with any bytes of the

checksum field or the data protected by the corresponding checksum field. With the well-formed sample and the malformed sample, CAFA only needs to run the program twice to identify the checksum point.

TaintScope treats all bytes read from the file as the taint source range, and this range is much larger than the range considered by CAFA. In addition, TaintScope leverages the symbolic execution and constraint solver to fix the checksum field of the crashed samples, that is, time consuming. CAFA can leverage the known checksum algorithm to fix the crashed samples simply and quickly.

5.2.2. Identification Accuracy. Table 6 lists the checksum points identified by specifying the checksum field as the taint source under the Taint-S(C) strategy. The fourth column shows the number of conditional jump instructions identified using the H1 and H2 rules, and the fifth column shows the number of conditional jump instructions identified using the H3 rule. Except for the rar application, all tested applications are open-source. The source line corresponding to the checksum point can be located based on the symbol information. By manually reviewing the nearby source code, we can easily confirm that there are no false positives for these tested applications. There are multiple types of checksum verification mechanisms in some file specifications. For example, there are two types of checksum verification mechanisms (i.e.,

TABLE 6: Checksum points.

Input Specification	Application	Checksum algorithm	cjl by H1 and H2	cjl by H3	Checksum point (assembly)	Checksum point (source)
PNG	ImageMagick	CRC32	57	1	at offset 0x7972 in libpng12.so.0.46.0 jz short loc_799D	at pngutil.c:167 if(png_crc_error(png_ptr))
		Adler32	39	1	at offset 0x922e in libz.so.1.2.3.4 je loc_A6B4	at inflate.c:1170 if ((REVERSE(hold)) != state->check) {
	optipng	CRC32	6	1	at offset 0x7972 in libpng12.so.0.46.0 jz short loc_799D	at pngutil.c:167 if(png_crc_error(png_ptr))
		Adler32	42	1	at offset 0x922e in libz.so.1.2.3.4 je loc_A6B4	at inflate.c:1170 if ((REVERSE(hold)) != state->check) {
	pngcheck	CRC32	5	1	at offset 0x11df6 in pngcheck jz short loc_8059E50	at pngcheck.c:4594 if (filecrc != CRCCOMPL(crc)) {
		Adler32	42	1	at offset 0x922e in libz.so.1.2.3.4 je loc_A6B4	at inflate.c:1170 if ((REVERSE(hold)) != state->check) {
GZIP	gzip	CRC32	28	1	at offset 0x978e in gzip jz short loc_80517B8	at unzip.c:185 if (orig_crc != updcrc(outbuf, 0)) {
ZIP	unzip	CRC32	8	1	at offset 0x70d6 in unzip jz loc_804F1B6	at extract.c:1978 if (G.crc32val != G.lrec.crc32) {
TAR	tar	Tar checksum	12	2	at offset 0x16eae in tar jz short loc_805EEB8	at list.c:352 if (unsigned_sum != recorded_sum && signed_sum != recorded_sum)
RAR	rar	CRC32	6	1	at offset 0xe728 in rar jz loc_8055EB0	—
UDP	tcpdump	UDP Checksum	10	1	at offset 0x73115 in tcpdump jz loc_80BB99F	at print-udp.c:562 if (sum != 0) {
TCP	tcpdump	TCP Checksum	8	1	at offset 0x6e68b in tcpdump jz loc_80B7B0C	at print-tcp.c:390 if (sum != 0)
IP	tcpdump	IP Checksum	10	1	at offset 0x31615 in tcpdump jz loc_807932D	at print-ip.c:634 if (sum != 0) {
IGMP	tcpdump	IGMP Checksum	8	1	at offset 0x30146 in tcpdump jz loc_80780C3	at print-igmp.c:335 if (in_cksum(vec, 1))

CRC32 and Adler32) in the PNG specification. In order not to miss any type of the checksum point, we need to construct a malformed sample that destroys the corresponding checksum field for each type of the checksum verification mechanism. All types of checksum verification mechanisms are specified in the file specification, so CAFA can identify all types of checksum points in principle. In other words, CAFA can eliminate false negatives in principle.

As the paper claimed, TaintScope could only locate the checksum point of the CRC32 type, missing the checksum point of the Adler32 type for Adobe Acrobat. Due to the CRC32 integrity check failure, Adobe Acrobat refused to decompress the image data in malformed Deflate-compressed PNG images and exited immediately. Because Adobe Acrobat is an application running on the Windows operating system, CAFA cannot test it. But in theory, CAFA is able to identify the two types of checksum points for Adobe Acrobat as follows. CAFA constructs the first malformed

sample by tampering with the CRC32 field to identify the checksum point of the CRC32 type, and patches the application at the checksum point. CAFA then constructs the second malformed sample by tampering with the Adler32 field. Since the Adler32 field is part of the calculation of the CRC32 checksum value, tampering with this field will make the second malformed sample unable to pass the CRC32 verification of the original application. But the second malformed sample can pass the CRC32 verification of the patched application. When the patched Adobe Acrobat decompresses the data of the second malformed sample, CAFA can identify the checksum point of the Adler32 type.

On the Ubuntu operating system, we have tested similar applications (such as optipng, ImageMagick, and pngcheck) that are also able to parse Deflate-compressed PNG files. In the Deflate-compressed PNG file, one IDAT chunk contains data that is Deflate-compressed in the zlib format (see Table 7). The check value stored at the end of the zlib

TABLE 7: Zlib format.

Name	Size
Compression method/flags	1 bytes
Additional flags/check bits	1 bytes
Compressed data blocks	n bytes
Check value	4 bytes

data stream is calculated by the Adler32 algorithm on the uncompressed data. For example, when the optipng application parses the Deflate-compressed PNG file, two steps of checksum verification including the CRC32 verification and the Adler32 verification are required to check the integrity of the IDAT chunk. The experimental results show that CAFA can identify the two types of checksum points for optipng, ImageMagick and pngcheck (see Table 6).

5.3. Coverage Statistics. To evaluate the effect of CAFA on the coverage, we compared the coverage of AFL (baseline) with the coverage of AFL assisted by CAFA. The coverage is measured by counting the number of new edges explored during each test. After fuzzing the ImageMagick and optipng applications for two hours, we obtained the coverage graphs shown in Figure 5. These graphs show that the coverage of AFL is greatly improved by CAFA. The coverage increased by 9.1x when testing the optipng application and by 15.4x when testing the ImageMagick application. Because most samples generated by AFL failed to pass the checksum verification in the original application, the coverage of AFL (baseline) was very low. However, most samples generated by AFL could bypass the checksum verification with the assistance of CAFA, so the coverage was significantly improved.

In addition, we compared CAFA with AFLfast [25], a popular open-source extension of AFL. As shown in Figure 5, the coverage of AFLfast is not even as good as that of AFL (baseline) for the tested applications. Therefore, AFLfast does not help to improve the coverage when testing an application that uses the checksum mechanism.

5.4. Fuzzing Results. The application is only statically patched at the checksum point and remains the same elsewhere, so CAFA is compatible with most fuzzing tools. For the implementation, TaintScope patched an application by means of dynamic instrumentation and used its own fuzzing engine, making it incompatible with other fuzzing tools.

AFL can test more code with the help of CAFA, so the probability of finding vulnerabilities is increased. After the program is anti-logically patched, most of the malformed samples generated by the fuzzer follow the passing verification branch to test the deeper code. All of the code except the conditional jump instruction at the checksum point is unchanged, so there is no risk of overwriting some vulnerabilities due to patching. Conversely, if the fuzzer directly comments out the checksum verification code (AFL) or forces the program to follow the passing verification branch (TaintScope), some vulnerabilities in the no-passing verification branch may be missed.

FIGURE 5: Coverage graphs.

With the help of CAFA, we found several known security vulnerabilities (see Table 8) in the legacy ImageMagick application (version 7.0.6-1). These vulnerabilities were located after the checksum verification and it was very difficult to pass the checksum verification for the samples generated by AFL, so it was difficult to find these vulnerabilities just relying on AFL. In addition, we found fourteen crashed samples for the latest version of the rar application (version 1.26) and three crashed samples for the latest version of the ImageMagick application (version 7.0.7-2). Through manual debugging analysis, we found that these crashed samples only caused the application to denial of service, but could not be further exploited for remote code execution.

6. Related Work

CAFA builds upon the taint analysis technique to help fuzzing tools bypass the checksum verification to achieve higher test coverage. In this section, we review the existing methods of improvement to fuzzing tools for increasing the test coverage.

TABLE 8: Security vulnerabilities of the imagemagick application.

CVE ID	Vuln-Type	Severity Rating
CVE-2017-17879	heap-based buffer over-read	Medium
CVE-2017-17504	heap-based buffer over-read	Medium
CVE-2017-12641	memory leak	High
CVE-2017-11522	NULL pointer dereference	Medium
CVE-2017-12643	memory exhaustion	High
CVE-2017-12435	memory exhaustion	High

6.1. White-Box Fuzzing. Some systems [2–4, 26] use symbolic execution techniques to generate inputs by solving constraints to help the fuzzing tools test more paths. For example, Driller [26] attempts to leverage selective concolic execution to help AFL satisfy the complex checks separating compartments. However, the application of Driller to large, real-world applications remains challenging. The current symbolic execution engines and constraint solvers cannot accurately generate and solve the constraints that describe the complete process of checksum verification algorithms.

6.2. Grammar-Based Fuzzing. Grammar-based fuzzing uses a specific grammar to generate the well-formed samples to test the program. The grammar that delineates the format information is usually constructed manually. If the input specification is unknown, it needs a lot of time to reverse the program. Even if the input specification is public, it will take some time to construct a grammar that meets the requirements of the fuzzing tool, such as the peach pit [10].

Jingbo Yan [11] presented a novel approach for fuzzing highly structured input programs. They inferred the grammar structures by disassembling the existing test cases into multiple grammatical fragments. YMIR [13] uses API-level concolic testing to construct the grammar automatically for ActiveX control, and Godefroid et al. [14] automated the generation of the PDF grammar using neural-network-based statistical machine-learning techniques. However, these approaches are inaccurate and are time-consuming.

Grammar-based fuzzing is slower than CAFA because it needs to calculate the checksum value for each mutated sample based on the grammar. In addition, this approach cannot test the paths that do not pass the checksum verification.

6.3. Coverage-Based Greybox Fuzzing. Coverage-based grey-box fuzzing (CGF) is a random testing approach that records coverage information using lightweight instrumentation. If a mutated sample generated by the fuzzer exercises a new and interesting path, the fuzzer retains this sample. AFL is one of the most popular CGF techniques, and a large number of vulnerabilities have been found by it. Researchers have presented various improvements to AFL in recent years that have achieved good results. VUzzer [27] assigns weights to basic blocks based on control-flow features to prioritize and deprioritize certain paths. AFLGo [28] attempts to direct the grey-box fuzzing process to explore specific paths with the objective of reaching a target location and uses simulated annealing as a practical global meta-heuristic during

test generation. AFLfast [25] treats coverage-based grey-box fuzzing as the exploration of the state space of a Markov chain, and forces AFL to visit more states that are hidden in low-density regions. However, the above improved methods are essentially dependent on random mutation. It is almost impossible for the random mutation to generate well-formed samples satisfying the complex checksum verification, so these methods often achieve low coverage for applications with checksum verification mechanisms.

6.4. Checksum-Aware Directed Fuzzing. TaintScope [8] is a Checksum-Aware Directed Fuzzing tool that can automatically identify the checksum points based on the taint analysis technique. It forces the program to execute the passing verification branch by dynamic instrumentation at the checksum point and employs the STP constraint solver to generate the checksum value to repair a crashed sample. Compared with TaintScope, CAFA takes full advantage of the public file specifications and has the following advantages. (1) CAFA identifies checksum points much faster than TaintScope (see Section 5.2.1). (2) CAFA can identify the nested checksum points that TaintScope cannot identify (see Section 5.2.2). (3) CAFA is more general and can be compatible with more fuzzing tools than TaintScope (see Section 5.4). (4) CAFA is able to repair crash samples more quickly using the checksum algorithm defined in the input specification. But TaintScope takes more time to generate checksum values through the STP constraint solver, and some constraints cannot be solved.

7. Conclusions and Future Work

In this paper, a checksum-aware fuzzing assistant tool (CAFA) is designed to identify the checksum point automatically based on taint analysis. CAFA can quickly and accurately identify the checksum point, and significantly improve the coverage of AFL. By testing deeper code with the help of CAFA, we have found deep vulnerabilities that could not be found only by AFL.

However, there are some limitations to the application of CAFA as follows. (1) The CRC32-S strategy can only be used for the application that uses the crc32 function of the libz.so library to implement the checksum algorithm. (2) The Taint-S strategy can be used when the location of the checksum field or the data protected by the corresponding checksum field in the sample is known. (3) To fix the checksum field of the crashed sample, the checksum algorithm needs to be known. In summary, to use CAFA, we only need to find the part

related to the checksum verification from the public input file specification. In fact, many input specifications are public, such as PNG, GZIP, RAR, and TCP.

Occasionally, fuzzing tools stumble not due to the checksum but due to some other calculation that depends on multiple elements of the input. Through taint analysis, we can determine which input bytes influence the behavior of the conditional jump instructions. In the future, CAFA can use the information gathered through dynamic analysis to guide the fuzzing tool to change specific bytes, so that both branches of the conditional jump instruction can be tested.

Data Availability

The open-source code and samples used to support the findings of this study have been deposited in the GitHub repository (<https://github.com/CAFA1/CAFA.git>).

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

This work was supported by the Ministry of Science and Technology of China under Grant no. 2017YFB0802901.

References

- [1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
- [2] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: whitebox fuzzing for security testing: SAGE has had a remarkable impact at Microsoft," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.
- [3] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: a binary analysis platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, pp. 463–469, Snowbird, UT, USA, 2011.
- [4] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX USENIX Conference on Operating Systems Design and Implementation*, pp. 209–224, San Diego, 2008.
- [5] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.
- [6] X. Wang, L. Zhang, and P. Tanofsky, "Experience report: How is dynamic symbolic execution different from manual testing? A study on KLEE," in *Proceedings of the 24th International Symposium on Software Testing and Analysis, ISSTA 2015*, pp. 199–210, USA, July 2015.
- [7] lcamtuf, "American fuzzy lop," <http://lcamtuf.coredump.cx/afl/>.
- [8] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP '10)*, pp. 497–512, IEEE Computer Society, May 2010.
- [9] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by Program Transformation," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, pp. 697–710, San Francisco, CA, May 2018.
- [10] "Peach fuzzer: discover unknown vulnerabilities," 2018, <https://www.peach.tech/>.
- [11] J. Yan, Y. Zhang, and D. Yang, "Structurized grammar-based fuzz testing for programs with highly structured inputs," *Security and Communication Networks*, vol. 6, no. 11, pp. 1319–1330, 2013.
- [12] K. Serebryany, "Continuous Fuzzing with libFuzzer and AddressSanitizer," in *Proceedings of the 2016 IEEE Cybersecurity Development, SecDev 2016*, p. 157, USA, November 2016.
- [13] S. Y. Kim, S. Cha, and D.-H. Bae, "Automatic and lightweight grammar generation for fuzz testing," *Computers & Security*, vol. 36, pp. 1–11, 2013.
- [14] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pp. 50–59, USA, November 2017.
- [15] "RFC 1180: A TCP/IP Tutorial," 2018, <https://www.rfc-editor.org/info/rfc1180>.
- [16] "PNG (Portable network graphics) specification," 2018, <http://www.libpng.org/pub/png/spec/1.2/png-1.2-pdg.html>.
- [17] "MNG1.0," 2018, <http://www.libpng.org/pub/mng/spec/>.
- [18] "RFC 1952 GZIP file format specification version 4.3," 2018, <http://www.zlib.org/rfc-gzip.html>.
- [19] "RAR 5.0 archive format," 2018, <https://www.rarlab.com/technote.htm>.
- [20] "ZIP file format specification," 2018, <https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT>.
- [21] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 ACM International Symposium on Software Testing and Analysis (ISSTA '07) and PADTAD-V Workshop*, pp. 196–206, London, UK, July 2007.
- [22] "010 editor - binary template repository - download binary templates," 2018, <https://www.sweetscape.com/010editor/repository/templates/>.
- [23] "Wireshark go deep," 2018, <https://www.wireshark.org/>.
- [24] C.-K. Luk, R. Cohn, R. Muth et al., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pp. 190–200, June 2005.
- [25] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security, CCS 2016*, pp. 1032–1043, Austria, October 2016.
- [26] N. Stephens, J. Grosen, C. Salls et al., "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, Calif, USA, 2016.
- [27] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, Calif, USA, 2017.
- [28] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference*, pp. 2329–2344, Dallas, Texas, USA, October 2017.