IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 24, 2019, accepted February 10, 2019, date of publication February 27, 2019, date of current version March 18, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2901951

Automatically Patching Vulnerabilities of Binary Programs via Code Transfer From Correct Versions

YIKUN HU, YUANYUAN ZHANG, AND DAWU GU
School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China

Corresponding author: Dawu Gu (dwgu@sjtu.edu.cn)

This work was supported in part by the Key Program of National Natural Science Foundation of China under Grant U1636217, in part by the National Key Research and Development Program of China under Grant 2016YFB0801201 and Grant 2016QY071401, and in part by the Major Project of Ministry of Industry and Information Technology of China under Grant [2018] 282.

ABSTRACT The security of binary programs is significantly threatened by software vulnerabilities. When vulnerabilities are found, those applications are exposed to malicious attacks that exploit the known vulnerabilities. Thus, it is necessary to patch them when vulnerabilities are reported to the public as soon as possible. However, it still heavily relies on manual work to locate and correct the corresponding defective code in the binary programs. In order to raise productivity and ensure software security, it becomes imperative to automate the process. In this paper, we propose BINPATCH to automatically patch known vulnerabilities of binary programs. It first locates the defective function, which contains the vulnerability, via similar code comparison. Then, it reuses the corresponding code from the correct version of the defective function as the patch code and inserts it to the defective function via binary rewriting. BINPATCH is evaluated on eight real-world vulnerabilities, and the experimental results show that it is able to not only locate the defective code effectively but also patch the code correctly.

INDEX TERMS Reverse engineering, binary code patching, binary program analysis, software security.

I. INTRODUCTION

Vulnerabilities constitute one of the largest threats to the software security. Although developers and analyzers have spent much effort to remove the vulnerabilities, many of them, even including security-critical ones, still exist in the released software, i.e., binary programs or executables. In addition to the potential vulnerabilities which have not been reported (i.e., zero-day vulnerabilities), binary programs are even threatened by malicious attacks exploiting *known* ones. When a vulnerability is found and reported to the public, vendor cannot always provide the corresponding patch for their released binaries in time, let along for legacy binary applications which lack regular maintenance. It has been found that many vulnerabilities have the life span over 12 months, generalized across numerous types of software [38]. Therefore, in order to ensure the security, it is necessary to patch the binary programs under that situation. However, the process stills heavily relies on manual work which is tedious and time-consuming.

The associate editor coordinating the review of this manuscript and approving it for publication was Ning Weng.

To automatically patch the binary programs, there exist two challenges to be handled. The first one is defective code localization (**C1**). Released binary programs commonly have been *stripped* that their debug information and symbol names are discarded. Even though the vulnerabilities are known, it is difficult to find the corresponding defective code in stripped binaries. The other challenge is to generate correct and compatible patch code which protects the defective binary code from vulnerabilities (**C2**). Binary code (or machine code) is a low-level language which needs to manage the memory, including allocating registers, arranging the stack and heap, etc. After the compilation, the memory management of binary code has been completed by compilers. If the inserted patch code occupies or modifies the managed memory improperly, the resulting binary code would produce faulty results, e.g., segmentation faults. Thus, the generated patch code is required to not only correct the defective code, but also avoid breaking the context of the original program.

In the literature, EC [56] is the typical solution for binary program patching. On one hand, it locates the defective code with the spectrum-based fault localization technique, which requires numerous test cases to fulfill the target [66].

Nevertheless, binary programs, especially the legacy ones, usually lack test cases. Actually, it is still an open question for testing techniques to trigger arbitrary code and obtain full coverage of binaries [32]. On the other hand, EC generates candidate patch code by manipulating the defective code with empirical rules (including instruction insertion, removal, and swapping), then validates the candidates with test cases. Because of the pre-defined manipulations, EC lakes specifications of intended behaviors. It adopts test cases as the criteria to search for the real patch code. As a result, it would produce plausible patches [41] which pass all tests but do not actually remove the vulnerabilities. ClearView [54] and SafeStack [10] monitor the running binary programs and patch the vulnerabilities online once they find erroneous behaviors. The two solutions are short-term, and they both bring extra overhead to the original execution of programs. KARMA [11] focuses on patching Android kernels which are not stripped, i.e. the debug and symbol information is reversed. Therefore, it is less effective to handle COTS (Commercial Off-The-Shelf) binaries which are commonly stripped.

In this paper, we propose BINPATCH, an automatic methodology patching known vulnerabilities of binary programs. Given the defective function that contains the known vulnerability, BINPATCH firstly locates the corresponding function in the target binary program via code similarity comparison. Then, it extracts the code from the correct version of the defective function as the patch code. BINPATCH transforms the patch code into compatible form, and inserts it into the target program via binary rewriting.

Since the vulnerability has been known, in order to overcome **C1**, BINPATCH adopts the technique of binary code similarity comparison which is effective in locating known vulnerabilities [9], [17], [20], [21], [55]. The technique could handle stripped binaries and has no need for test cases. To solve **C2**, BINPATCH extracts the patch code from the correct version of the defective function, ensuring the patch could handle the vulnerability correctly. It does not rely on test cases to specify the intended behaviors of the patch code. Besides, BINPATCH performs transformation on the extracted code to make it compatible with the target program. We adopt eight real-world vulnerabilities to evaluate the capacity of BINPATCH. The results show that it is able to not only locate defective code effectively, but also patch the vulnerabilities correctly.

In summary, the contributions of this paper are as followed:

- We propose an automatic method to patch known vulnerabilities of binary programs. With the code of a known vulnerability, we locate the defective function in the binary program via code similarity comparison, and generate patch code by reusing corresponding code from the correct version of the defective function.
- We propose the technology to fulfill patch code extraction and transfer between binary programs. We align the (emulated) execution traces of the defective function and its correct version to extract the patch code. After the processes of Used Register Protection, Function Argument Replacement, and Conditional Jump Targets Determination, the patch code is then transformed into the compatible form for the target defective function. The method is capable of handling vulnerabilities which require complex operations to patch, and avoids the overfitting caused by test case validation.
- We implement the approach in a prototype system named BINPATCH. We evaluate BINPATCH on eight real-world vulnerabilities. The experimental results show that BINPATCH is able to locate the defective code effectively and patch it correctly.

II. MOTIVATION AND OVERVIEW

In this section, we use an example to illustrate the challenges of patching vulnerabilities in binary programs and explain the basic idea of BINPATCH.

A. MOTIVATING EXAMPLE

`NConvert`[1] is a closed-source image processor for multiple file formats. It adopts the open-source library `libpng`[2] to handle PNG (Portable Network Graphics) files. According to the change log, before the version of 6.82, NConvert links the *static* libpng which contains an buffer overflow (CVE-2015-8126).[3] The vulnerability was reported in Nov. 12, 2015, while NConvert was not updated until Apr. 1, 2016 when v6.82 was released. It replaced the vulnerable libpng v1.6.18 with the correct version of 1.6.20. Listing 1 shows the source and assembly code of the defective function *png_set_PLTE*. It lacks the length limitation for the PLTE chunk (`num_palette` at Line 10), allowing attackers to cause a denial of service with a crafted image. Listing 2 presents the correct version of the function. Each '+' sign marks a new line added by the patch, and '-' represents the removed statement. The patch code defines a new variable `max_palette_length` to ensure the legality of the PLTE chunk length (Lines 9-12 and 16, Line 33-39 and 43-44).

The first challenge for the patching is defective code localization (**C1**). We download NConvert v6.17 from its home page, disassemble it with `IDA Pro` v6.6,[4] and manually locate *png_set_PLTE* at address `0x81AE780`. We also find its caller functions which invoke subroutines explicitly in NConvert, as shown in Figure 1. Each node represents a function, and every directed edge points from a caller to its callee. Hex numbers are function start addresses in NConvert. By manual analysis, we find corresponding statically-linked libpng functions and attach their names in the figure. *func_80be580* and *func_8139cd0* are two user-defined functions of NConvert.

[1] https://www.xnview.com/en/nconvert
[2] http://www.libpng.org/pub/png/libpng.html
[3] https://www.cvedetails.com/cve/CVE-2015-8126
[4] https://www.hex-rays.com/products/ida/index.shtml

```
1  void png_set_PLTE(png_structrp png_ptr,
2      png_inforp info_ptr, png_const_colorp palette,
3      int num_palette){
4
5    if (png_ptr == NULL || info_ptr == NULL)
6       return;
7
8    // lacking limitation for over-length PLTE chunk
9    if (num_palette < 0 ||
10      num_palette > PNG_MAX_PALETTE_LENGTH){
11      ...
12   }
13   ...
14 }
```

```
15   sub     esp, 3Ch
16   mov     [esp+3Ch+var_C], esi
17   mov     esi, [esp+3Ch+arg_4] ; esi = info_ptr
18   mov     [esp+3Ch+var_8], edi
19   mov     edi, [esp+3Ch+arg_0] ; edi = png_ptr
20   test    esi, esi ; if (info_ptr == NULL)
21   mov     [esp+3Ch+var_4], ebp
22   mov     ebp, [esp+3Ch+arg_C] ; ebp = num_palette
23   jz      short loc_1FEBB
24   test    edi, edi ; if (png_ptr == NULL)
25   jz      short loc_1FEBB ; if (num_palette
26   cmp     ebp, 100h       ;    == PNG_MAX_PALETTE_LENGTH)
27   ja      loc_1FED0
28   ...
```

Listing 1. Defective code of libpng v1.6.18.

There are no existing test cases for NConvert. If locating *png_set_PLTE* in it with test cases (e.g., EC [56]), we need to construct crafted inputs to trigger one of its callers firstly. Nevertheless, *func_8139cd0* has no explicit callers in NConvert. It might be invoked via indirect calls, and unfortunately, its callers are difficult to decide accurately with existing methods [47]. *png_read_png* has no caller either. It might be just statically-linked but never invoked. The remaining function *func_80be580* is the subroutine of another 21 NConvert functions. Then we need to further analyze NConvert in order to find possible execution paths for the defective function, which is still an issue of binary program analysis [32].

The second challenge is the generation of correct and compatible patch code (**C2**). The addition and removal of code presented in Listing 2 could be considered as the best practice and baseline of patching. It leverages the branch structure to decide the value of `max_palette_length` (Line 9-12, Line 33-39) which is then compared with `num_palette` (Line 16, Line 43-44). Since the above operations are completely new to the original function, it is difficult to generate them with pre-defined instruction manipulation rules.

B. SYSTEM OVERVIEW

For binary program patching, we propose BINPATCH to handle the challenges. Figure 2 presents the work flow of BINPATCH. In the first step of Defective Function Localization (§III-A), given the defective binary program NConvert v6.17 which contains the defective function *png_set_PLTE*, BINPATCH locates the defective function via binary code similarity comparison. With the source code of libpng v1.6.18, we compile it and set the binary code of

```
1  void png_set_PLTE(png_structrp png_ptr,
2      png_inforp info_ptr, png_const_colorp palette,
3      int num_palette){
4  +   png_uint_32 max_palette_length;
5
6    if (png_ptr == NULL || info_ptr == NULL)
7       return;
8
9  +   max_palette_length =
10 +     (info_ptr->color_type == PNG_COLOR_TYPE_PALETTE) ?
11 +     (1 << info_ptr->bit_depth) :
12 +     PNG_MAX_PALETTE_LENGTH;
13
14   if (num_palette < 0 ||
15 -     num_palette > PNG_MAX_PALETTE_LENGTH)
16 +     num_palette > (int) max_palette_length)
17      ...
18   }
19   ...
20 }
```

```
21   push    ebp
22   push    ebx
23   push    edi
24   push    esi
25   sub     esp, 0Ch
26   mov     esi, [esp+1Ch+arg_0] ; esi = png_ptr
27   test    esi, esi ; if (png_ptr == NULL)
28   jz      loc_20366
29   mov     ebp, [esp+1Ch+arg_4] ; epb = info_ptr
30   test    ebp, ebp ; if (info_ptr == NULL)
31   jz      loc_20366
32   mov     edi, [esp+1Ch+arg_C] ; edi = num_palette
33 + mov     al, [ebp+19h] ; al = info_ptr->color_type
34 + mov     edx, 100h ; edx = PNG_MAX_PALETTE_LENGTH
35 + cmp     al, 3           ; if (info_ptr->color_type ==
36 + jnz     short loc_202D5 ;    PNG_COLOR_TYPE_PALETTE)
37 + mov     cl, [ebp+18h] ; cl = info_ptr->bit_depth
38 + mov     edx, 1
39 + shl     edx, cl ; edx = 1 « info_ptr->bit_depth
40 loc_202D5:
41   test    edi, edi ; if (num_palette < 0)
42   js      loc_2036B
43 + cmp     edx, edi  ; if (max_palette_length <
44 + jl      loc_2036B ;    num_palette)
45   ...
```

Listing 2. Corrected code of libpng v1.6.20.

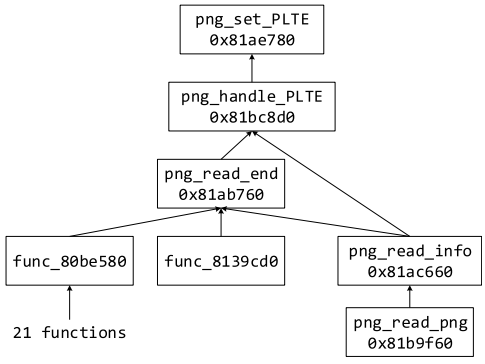


FIGURE 1. Call graph of *png_set_PLTE* in NConvert v6.17.

png_set_PLTE as the reference vulnerable function. Then, BINPATCH is adopted to compare the reference function to each target function of NConvert in pairs and calculate a similarity score. After this step, BINPATCH generates a list of candidate defective functions from NConvert, which is ranked by the similarity scores in descending order. It considers the function with the highest similarity score as the candidate to patch.

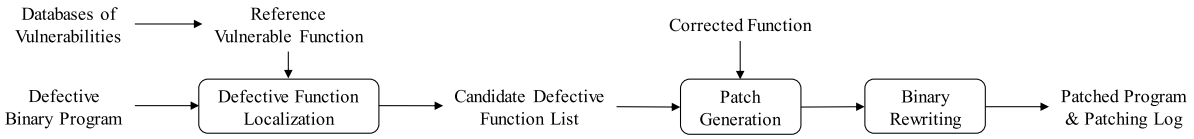


FIGURE 2. System architecture of BINPATCH.

Next, in the step of Patch Generation (§III-B), BINPATCH extracts the patch code from *png_set_PLTE* v1.6.20, the correct version. It runs instrumented *png_set_PLTE* v1.6.20 with the input which triggers the vulnerability, and emulates the candidate defective function located in NConvert with the same input. The (emulated) execution paths of the two functions are recorded simultaneously. Basing on the differences between the two paths, BINPATCH generates the patch code. In the last step of Binary Rewriting (§III-C), BINPATCH inserts the patch code into NConvert and outputs the patched binary program along with a log recording the rewriting information, including insertion point address, patch code size, etc.

In summary, BINPATCH focuses on patching known vulnerabilities in binary programs. To address **C1**, it adopts the technique of binary code similarity comparison. With reference vulnerable function, it is able to locate the defective function in the stripped binary program. To solve **C2**, it reuses the corresponding code from the correct version of the defective function as the patch code, which guarantees to generate patches with intended behaviors.

III. METHODOLOGY

In this section, we explain each step of BINPATCH and discuss how it patches known vulnerabilities of binary programs in details.

A. DEFECTIVE FUNCTION LOCALIZATION

BINPATCH is proposed to patch known vulnerabilities of binary programs. Thus, it locates the defective function in the target binary program by code similarity comparison. According to bug reports or databases of vulnerabilities, e.g., CVE (Common Vulnerabilities and Exposures),[5] analyzers obtain the information of the reference functions which contain the known vulnerabilities. Then, they use BINPATCH to locate the similar match of the reference function in the target defective binary program.

We base the process on `CACompare` [26], a semantics-based similar binary function detector. CACompare performs the similarity comparison via sampling. Given the reference function with the known vulnerability ($\mathcal{V}$), it provides $\mathcal{V}$ and every target function $\mathcal{T}$ of the defective program with identical random values as the input, and emulates their execution. The semantic signature of each function is captured during the emulation. Then, it compares the signature of $\mathcal{V}$ to that of each $\mathcal{T}$ in pairs, and calculates a score to measure their similarity. After this step, BINPATCH obtains the list of $\mathcal{T}$ which is sorted basing on the similarity scores in descending order. The Top *1* function on the list is considered as the candidate defective function to be patched.

[5]https://cve.mitre.org/

B. PATCH GENERATION

In this step, BINPATCH extracts the patch code ($\mathcal{P}$) from the correct function ($\mathcal{C}$). It firstly captures the path conditions of $\mathcal{C}$ and the candidate defective function $\mathcal{D}$, denoted as $P_{\mathcal{C}}$ and $P_{\mathcal{D}}$ (§III-B1). Then, it aligns $P_{\mathcal{C}}$ and $P_{\mathcal{D}}$ (§III-B2). The insertion point of $\mathcal{P}$ in $\mathcal{D}$ is decided as well. Basing on the results of alignment, $\mathcal{P}$ is extracted and transformed into the compatible form for $\mathcal{D}$ (§III-B3).

1) PATH TRIGGERING

BINPATCH executes the instrumented $\mathcal{C}$ with the test case that triggers the vulnerability. It records the *path condition* of the triggered path. Meanwhile, it records the runtime input values of $\mathcal{C}$, including argument values, accessed global variable values, heap variable values, and return values of subroutines [80]. Since the vulnerability is known, it is reasonable that the test cases covering the vulnerability are available. For example, such test cases of *png_set_PLTE* could be found in the CVE database. It is worth nothing that the test case is not to specify or validate the patch code, but to trigger the path which corresponds to the vulnerability.

Next, BINPATCH emulates $\mathcal{D}$ with the input values of $\mathcal{C}$, and captures the corresponding *path condition*. When $\mathcal{D}$ accesses global variables or heap variables, BINPATCH assigns corresponding ones of $\mathcal{C}$ basing on the accessing order. It is fulfilled in the same way to assign return values of subroutines. BINPATCH stops the emulation if $\mathcal{D}$ accesses illegal addresses, e.g., the overwritten return address caused by buffer overflow.

Figure 3 presents the extracted path conditions of *png_set_PLTE* in libpng v1.6.20 and NConvert v6.17 separately. In each sub-figure, the executed code is shown on the left. The instructions of conditional branches are marked with dotted boxes, from which the path conditions are inferred and shown on the right. For each entry on a line, the first tow elements are the variable values of the condition during the (emulated) execution. The third element is the condition flag which indicates how the two variable are compared (`Z`: zero, `L`: signed less than, `BE`: unsigned below or equal to). The last one is the corresponding result of the condition.

2) PATH CONDITION ALIGNMENT

In this step, BINPATCH aligns $P_{\mathcal{C}}$ and $P_{\mathcal{D}}$ to extract the patch code. Since the input is error-triggering, the unique conditions

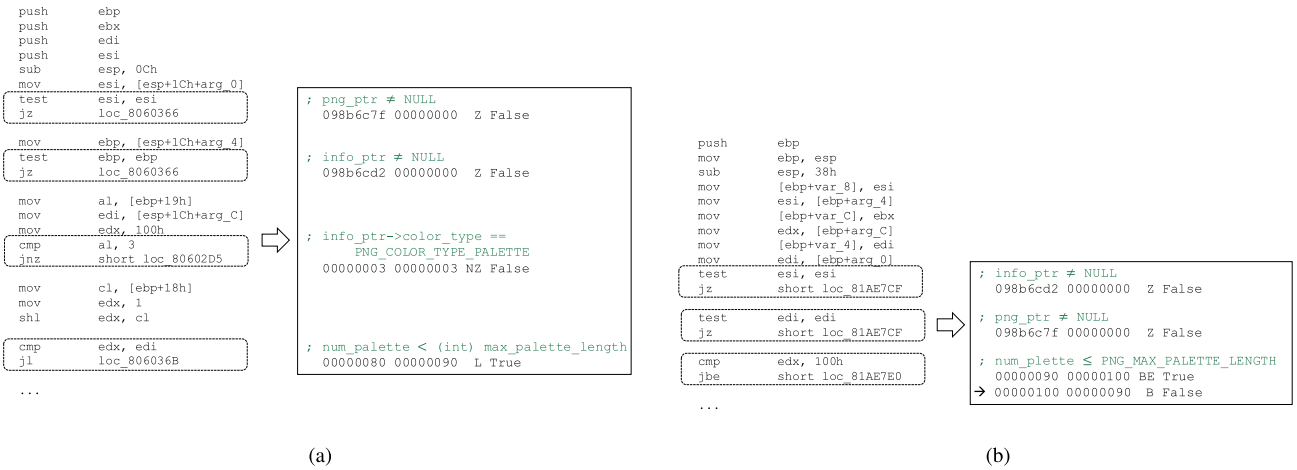


FIGURE 3. Path condition extraction. The left is the executed path, and the corresponding path condition is on the right. The variables are represented with the (emulated) runtime values. (a) libpng 1.6.20. (b) NConvert 6.17.

of $P_{\mathcal{C}}$ but missing in $P_{\mathcal{D}}$ indicate the code which removes the vulnerabilities.

Before the alignment, BINPATCH firstly pre-processes each path condition in order to ease the following alignment. Algorithm 1 presents the pseudocode. BINPATCH unifies all conditions along the triggered path into the strict inequalities with equivalent semantics (Line 4-5), i.e., strictly less than ($<$). For example, the original condition flag of the last condition in Figure 3b is `BE`. BINPATCH then unifies it with `B` that the variables are exchanged and the result is negated as well. Additionally, for each variable in the condition, BINPATCH captures all its possible values via value set analysis [4], [84]. Figure 4 shows the value sets of corresponding condition variables in Figure 3. `arg_`$\star$ represents the arguments of the function.

Algorithm 1 The Pre-Process of Path Condition

Input: Triggered Path Condition P
Input: Control Flow Graph of the Function G
Output: Path Condition P'
Output: Compound Logic L of the Path Condition
Output: Results of Value Set Analysis R
1 **Algorithm** *process_path_condition (P, G)*
2 $P' \leftarrow P$
3 $R \leftarrow \varnothing$
4 **foreach** *condition C along* P' **do**
5 $C \leftarrow$ **normalize_condition**(C)
6 **foreach** *variable V of C* **do**
7 $R \leftarrow R +$ **value_set_analysis**(V)
8 $L \leftarrow$ **identify_compound_logic**(P, G)
9 **return** P', L, R

Afterwards, BINPATCH identifies the compound logic operations in $P_{\mathcal{C}}$ and $P_{\mathcal{D}}$ separately (Line 8). For binary code, the implemented order of conditions in a compound logic might be different from that for the source code, which depends on the compilers and compilation settings,

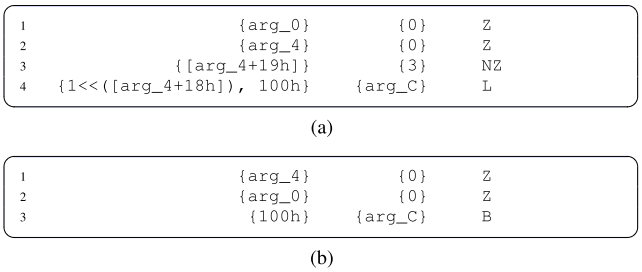


FIGURE 4. Value sets of condition variables. (a) libpng 1.6.20. (b) NConvert 6.17.

e.g., optimization options. Taking *png_set_PLTE* as an example, its disjunction

```
png_ptr == NULL || info_ptr == NULL
```

is implemented in the reverse order in Figure 3b, i.e., `info_ptr` is checked firstly, then `png_ptr`. In contrast, it is in the normal order in Figure 3a. Thus, when aligning conditions of compound logics, BINPATCH merely considers the contents, but ignores the order of conditions in the compound logic. Figure 5 depicts the simplified control

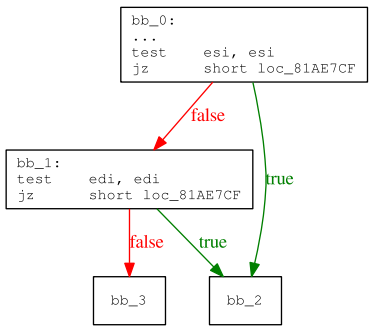


FIGURE 5. Simplified control flow graph of the disjunction in *png_set_PLTE* from NConvert 6.17.

flow graph (CFG) of that disjunction. Two conditions are considered to belong to a compound logic, if the basic blocks they reside in (`bb_0` and `bb_1`) satisfy: i) `bb_0` is the strict and immediate dominator of `bb_1`, and ii) `bb_0` and `bb_1` share one same immediate successor (`bb_2`). It is in the same way to decide whether two compound logics constitute a larger one.

Next, BINPATCH aligns the value sets of $P_{\mathcal{C}}$ and $P_{\mathcal{D}}$ with the longest common subsequence algorithm (LCS). Algorithm 2 presents the pseudocode deciding whether two conditions could be aligned, which is the core of LCS. For the condition M of $P_{\mathcal{C}}$ and N of $P_{\mathcal{D}}$, if they both belong to compound logics, BINPATCH gets all the conditions of the compound logics (M' and N'). If the contents of M' and N' are equivalent, then they could be aligned (Line 2-6). M could be aligned to N when their condition flags are equivalent, and each variable value set of N is the subset of the corresponding one of M (Line 7-12). For example, Line 1-2 in Figure 4a, which constitute a compound logic, could be aligned to the first two lines in Figure 4b, while Line 3 is unaligned with no corresponding elements. Additionally, because the condition flag `L` is equivalent to `B` when the variables are positive, Line 4 in Figure 4a could be aligned to the third line in Figure 4b. The insertion point in $\mathcal{D}$ is the program point after the path condition whose corresponding condition in $\mathcal{C}$ is the last aligned condition before the first unaligned one. Thus, in Figure 3b, the insertion point of NConvert is the program point right after the second path condition.

Algorithm 2 Alignment Decision

Input: The Condition M of $P_{\mathcal{C}}$
Input: The Condition N of $P_{\mathcal{D}}$
Input: Compound Logic $L_{\mathcal{C}}$ of $\mathcal{C}$
Input: Compound Logic $L_{\mathcal{D}}$ of $\mathcal{D}$
Output: The Decision Result, True or False

```
1  Algorithm is_aligned (M, N)
2      if M belongs to a compound logic
3      and N belongs to a compound logic then
4          M' ← get_compound_logic(M, L_C)
5          N' ← get_compound_logic(N, L_D)
6          return is_aligned_compound(M', N')
7      if not M belongs to a compound logic
8      and not N belongs to a compound logic then
9          if M.cond_flag is equivalent to N.cond_flag
10         and N.left_value_set ⊆ M.left_value_set
11         and N.right_value_set ⊆ M.right_value_set then
12             return True
13     return False
```

3) PATCH CODE EXTRACTION AND TRANSFORMATION

After the process of alignment, BINPATCH extracts the patch code from the control flow graph of the correct function, which is dominated by the basic block of the first unaligned path condition and post-dominated by that of the next aligned condition. BINPATCH performs backward program slicing [82], [83] on each variable of the patch code in order to obtain complete functionality. Figure 6 displays the process. Figure 6a gives the first unaligned and next aligned path condition of correct *png_set_PLTE* (v1.6.20) in dotted boxes, i.e., the third and fourth condition in Figure 6a. The instructions of the corresponding program slices are in the bold format, which are extracted as the patch code (Figure 6b).

Then, BINPATCH transforms the patch code into compatible form for the candidate defective function $\mathcal{D}$, including the

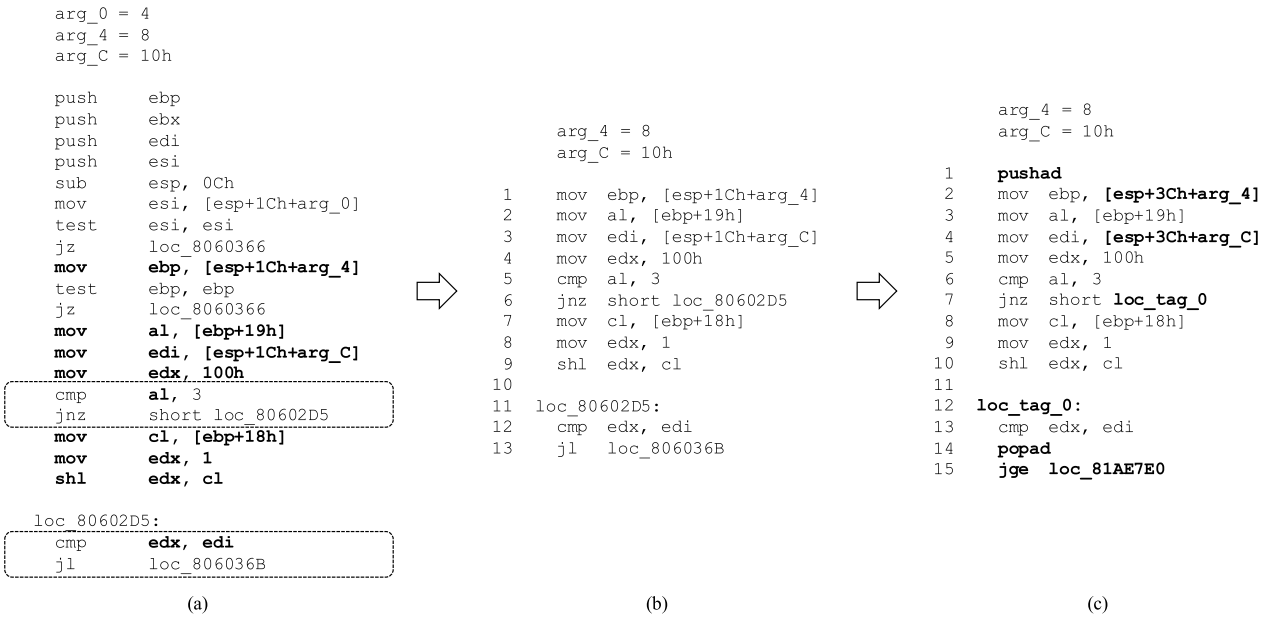


FIGURE 6. Patch extraction and transformation.

following processes: i) protecting used registers, ii) replacing function arguments, and iii) determining conditional jump targets.

Used Register Protection: Because of variant compilation settings, strategies that allocate registers are different. The patch code would occupy the registers of $\mathcal{D}$ and break the local context. Thus, BINPATCH saves the original register values onto the stack, and restores them before the exit of the patch code. As shown in Figure 6c, the instruction `pushad`, which pushes the contents of general-purpose registers onto the stack, is added at Line 1. `popad`, the reverse operation of `pushad`, is also added at Line 14 to restore the registers.

Function Argument Replacement: BINPATCH replaces the function arguments accessed in the patch code with corresponding ones of $\mathcal{D}$, because arguments on the stack might be accessed with different stack pointer registers (i.e., `esp` or `ebp`), which is decided by compilers. `ebp` would be used as a general-purpose register, as shown at Line 1 in Figure 6b, while that is rare for `esp`. BINPATCH then adopts `esp` to access arguments for the patch code. According to the calling convention, BINPATCH recognizes the arguments and obtains their offsets relative to `esp`. For example, the assembly function *png_set_PLTE* of NConvert in Figure 3b is compiled basing on *cdecl*, the default calling convention used by C compilers for the IA-32 architecture [22]. The arguments are prepared by the caller. Thus, for a callee, the arguments are placed on the high addresses relative to `esp` on the stack. At the very beginning, the compiler saves the value of `ebp` (4 bytes), then allocates a buffer of size `0x38` for local variables (`sub esp, 38h`), overall `0x3C` bytes ($= 4 + 0x38$). As a result, it is necessary to add `esp` with the offset of `0x3C` in order to access the memory field of arguments on the stack. The right operands at Line 2 and Line 4 in Figure 6c present the arguments after transformation.

Conditional Jump Targets Determination: Direct jump targets are implemented with hard-coding addresses in binary code, as shown at Line 6 and Line 13 in Figure 6b. BINPATCH needs to replace those address of the patch code with corresponding ones in $\mathcal{D}$. For jumps within the patch code (e.g., Line 6 in Figure 6b), their targets are decided by where the patch code is inserted in $\mathcal{D}$ in the step of Binary Rewriting (§III-C). Thus, they are represented with placeholders temporally (`loc_tag_0` at Line 7 in Figure 6c). For jumps between the patch code and the original code (e.g., Line 13 in Figure 6b), BINPATCH assigns their target values according to the results of path condition alignment (§III-B2). Because the condition at Line 12-13 in Figure 6b could be aligned to the last one in Figure 3b, the condition flag (`L`→`GE`) and target address (`loc_81AE7E0`) are assigned accordingly, as shown at Line 15 in Figure 6c.

C. BINARY REWRITING

BINPATCH inserts the patch code by adding a new `.text` section to the program, as shown in Figure 7. The left

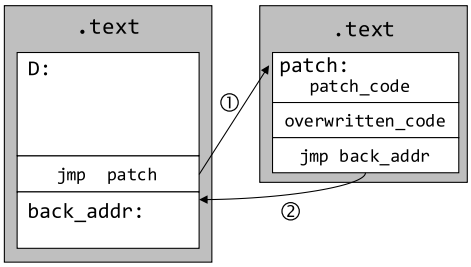


FIGURE 7. Binary program structure after rewriting.

`.text` is the original one containing the candidate defective function ($\mathcal{D}$), and the patch code is in the new `.text` on the right. At the insertion point, BINPATCH overwrites the original code with a jump instruction to the patch which generally consists of three parts (①). The first part is the compatible patch code (`patch_code`). The placeholders in the patch code are calculated by adding their offsets to the address where the patch is placed in $\mathcal{D}$ (`patch`). The second part is the normal code of $\mathcal{D}$ overwritten by the jump instruction (`overwritten_code`). The last part is the jump instruction back to $\mathcal{D}$ (②). After this step, BINPATCH generates the patched binary program. Besides, a log recording the information of the rewriting is generated as well, including insertion point address, overwritten code size, patch code size, etc.

IV. IMPLEMENTATION

BINPATCH supports binary code patching for ELF (Executable and Linkable Format) files on IA-32. Next, we introduce the solutions adopted in the implementation of BINPATCH.

A. BINARY INFORMATION EXTRACTION

BINPATCH requires argument offsets and function information (e.g., function addresses) to locate defective functions (§III-A). Processes of patch code generation (§III-B) and binary rewriting (§III-C) are also performed on assembly code. We leverage `IDA Pro`,[6] an industrial strength reverse engineering tool, to disassemble binary code. We develop scripts with its plugin `IDAPython` which provides API to disassemble binary code and extract the information automatically.

B. INSTRUMENTATION AND EMULATION

We adopt `Valgrind` [51] to implement the instrumentation of BINPATCH. We develop our own instrumentation plugin with API provided by Valgrind to record the path conditions and runtime information of the correct function (§III-B1). Valgrind adopts `VEX-IR`, a RISC-like intermediate representation, to unify the complex instruction set of IA-32. That feature facilitates the path triggering of BINPATCH.

[6]https://www.hex-rays.com/products/ida/index.shtml

TABLE 1. Objects of the evaluation.

Client Program	Library	CVE ID	Defective Version	Correct Version	Defective Function
GeoJasper	JasPer	CVE-2012-3352	1.900	1.900.2	jpc_dec_process_sot@jpc_dec.c:492
gif2tiff	libtiff	CVE-2013-4231	4.0.3	4.0.6	readraster@gif2tiff.c:336
Curl	libcurl	CVE-2016-7167	7.50.2	7.50.3	curl_easy_escape@escape.c:90
NConvert	libpng	CVE-2015-8126	1.6.18	1.6.20	png_set_PLTE@pngset.c:521
OpenSSL	libssl	CVE-2014-0160	1.0.1e	1.0.1g	tls1_process_heartbeat@t1_lib.c:2514
pngcheck	zlib	CVE-2005-2096	1.2.0.1	1.2.8	inflate_table@inftrees.c:129
tiffcp	libtiff	CVE-2006-2025	3.8.0	3.8.2	TIFFFetchData@tif_dirread.c:969
tiffinfo	libtiff	CVE-2010-2065	3.9.2	3.9.6	TIFFReadBufferSetup@tif_read.c:609

We implement the process of emulation with `angr` [62], a static binary analysis framework. It also bases on VEX-IR, and provides interfaces to simulate the execution of IR code. Furthermore, it has built in the module of value set analysis required in the step of path condition alignment (§III-B2).

V. EVALUATION

We evaluate BINPATCH with eight real-world Linux programs. Each one statically links a library which contains a known vulnerability in a defective function. The information of all the vulnerabilities is obtained from the CVE database.

A. EXPERIMENT SETUP

Table 1 lists the objects of the experiments. The first column (Client Program) presents the object program. The second column (Library) shows the library which is statically-linked in the client program. The third column (CVE ID) gives the CVE ID of each vulnerability. The fourth and fifth column list the defective and correct version of the library. The last column (Defective Function) gives the defective function and the line where the vulnerability occurs in the source code file. We disassemble binary code and identify functions with `IDA Pro` v6.6 (§IV-A). Besides, we instrument and capture runtime information of correct functions with `Valgrind` v3.12.0, and emulate the execution of the defective functions with `angr` (§IV-B). The experiments are conducted in Ubuntu 16.04 which runs on an Intel Core i5-2320 @ 3GHz CPU with 8G DDR3-RAM.

Ground Truth: BINPATCH performs patching on stripped binary programs whose debug and symbol information is discarded. To evaluate the effectiveness and capacity of BINPATCH, we compile copies of the object programs and libraries with the `-g` option to establish the *ground truth* with the debug information and symbol names. As the source code of `NConvert` is unavailable, we manually analyze the program via reverse engineering and find corresponding functions of `libpng` as references.

B. SUMMARY OF EXPERIMENTS

In the experiments, programs are compiled with their default configurations except for `NConvert` which is closed-source and downloaded from its homepage. Table 2 presents the summary of the experiments. The columns of Time show the time for patching in each step, and the last column (Condition Number) gives the number of path conditions for patch code generation. For all client programs, BINPATCH finishes the patching within 30 seconds. By manually verification, BINPATCH successfully locate the defective functions of all the client programs. It generates the correct patch code which protects each program from the vulnerability. Besides, the generated patch code certainly has no other side effects on original programs.

TABLE 2. Summary of BINPATCH patching results.

Client Program	Time (s)			Condition Number
	Defect Function Localization	Patch Extraction	Binary Rewriting	
GeoJasPer	9.478	0.149	0.003	1
gif2tiff	6.736	0.007	0.002	1
curl	1.026	0.151	0.002	1
NConvert	15.389	2.292	0.069	3
OpenSSL	9.673	0.019	0.001	2
pngcheck	9.237	2.684	0.038	1
tiffcp	5.314	2.136	0.002	3
tiffinfo	4.251	3.753	0.021	1

C. CASE STUDIES

We have taken `NConvert` as an example to illustrate how BINPATCH works in the section of Methodology (§III). Next, we discuss several other specific cases in details.

1) `OpenSSL` (CVE-2014-0160)

`Heartbleed` is a buffer over-read bug in `libssl` 1.0.1 through 1.0.1f. The Heartbeat Extension tests TLS secure communication links by allowing client to send a Heartbeat Request message, which consists of a payload and the payload's length. The server then must send back the same payload to the client. `OpenSSL` uses the function *tls1_process_heartbeat* to implement the process, as shown in Listing 3. When the server copies the message at Line 23, if it does not check the length to ensure whether the payload length (`payload`) is not greater than the actual length of the payload (`pl`), an attacker could send a small payload but with a large length so as to obtain sensitive data in the active

```
1 int tls1_process_heartbeat(SSL *s){
2    unsigned char *p = &s->s3->rrec.data[0], *pl;
3    unsigned short hbtype;
4    unsigned int payload, padding = 16;
5    /* Read type and payload length first */
6    hbtype = *p++;
7    n2s(p, payload);
8 +  if (1 + 2 + payload + 16 > s->s3->rrec.length)
9 +      return 0;
10   pl = p;
11   if (s->msg_callback) ...
12   if (hbtype == TLS1_HB_REQUEST){
13       ...
14       /* Allocate memory for the response,
15          size is 1 bytes message type,
16          plus 2 bytes payload length,
17          plus payload, plus padding */
18       buffer = OPENSSL_malloc(1+2+payload+padding);
19       bp = buffer;
20       ...
21       /* Enter response type, length and copy payload */
22       // size checking is required
23       memcpy(bp, pl, payload);
24       ...
25   }
26 }
```

Listing 3. Correct *tls1_process_heartbeat*.

```
1 int datasize, clear;
2 unsigned int prefix[4096];  // 4096 = (1 << 12)
3 unsigned char suffix[4096]; // 4096 = (1 << 12)
4 int readraster(void){
5     datasize = getc(infile);
6     clear = 1 << datasize;
7     ...
8     for (int code = 0; code < clear; code++){
9         prefix[code] = 0; // lacking checks for indexing
10        suffix[code] = code;
11    }
12    ...
13 }
```

Listing 4. Defective *readraster*.

```
1 char *curl_easy_escape(struct Curl_easy *data,
2     const char *string, int inlength){
3     size_t alloc =
4         (inlength?(size_t)inlength:strlen(string))+1;
5     ...
6     ns = malloc(alloc); // lacking ckecks for size
7     ...
8 }
```

Listing 5. Source code of defective *curl_easy_escape*.

memory of the server, which is contained in the responding message. The correct version of the function adds a sanity check to avoid Heartbleed, as shown at Line 8-9.

BINPATCH correctly locates the defective function *tls1_process_heartbeat* in `OpenSSL` v1.0.1e. When triggering the path condition, *tls1_process_heartbeat* v1.0.1g is executed with an erroneous input, and it returns directly because `payload` is overlarge. When emulating *tls1_process_heartbeat* v1.0.1e, the emulation stops at the point of memory copy (Line 23) because the process requires more contents (`payload`) than the payload (`pl`) actually has, and there is no enough legal data to provide for the emulation. Afterwards, BINPATCH generates the patch code which exits the function if the input fails the check, then inserts it into the defective function finally.

2) `gif2tiff` (CVE-2013-4231)

`gif2tiff` is a utility which converts GIF (Graphics Interchange Format) images to TIF (Tagged Image File Format) with the help of the library `libtiff`. When processing GIF images, it needs to iterates over the LZW (Lempel-Ziv-Welch) code size which should be less than `13`. The process is implemented with function *readraster*. However, libtiff v4.0.3 lacks the check that constrains the code size to be less than `13`, as presented in Listing 4. The `prefix` and `suffix` arrays are defined with a constant length `4096` (2^{12}) at Lines 2-3. If the `datasize` is over `12`, then the two arrays would be forced to overwrite a set of statically allocated buffers at Lines 9-10. libtiff v4.0.6 provides the patch condition before the indexing:

```
if (datasize > 12) return 0;
```

BINPATCH locates the defective *readraster* in gif2tiff v4.0.3 successfully. When performing the backward slicing on the path condition extracted from libtiff v4.0.6, it finds that `datasize` is the return value of `glibc` function *getc*, which is stored in `eax`:

```
; call    _IO_getc
  cmp     eax, 0Ch
  jg      loc_804A48D; the function epilogue
```

The jump target (`0x804A48D`) is the epilogue of the function which cleans the frame of the function stack and returns. BINPATCH then inserts the patch code to the defective *readraster* before the comparison of the loop and after the calling of *getc*, which is right before Line 8 in Listing 4 on the source code level.

3) `CURL` (CVE-2016-7167)

`Curl` is a tool for getting and sending files using URL syntax. Curl v7.50.2 suffers from an integer overflow in function *curl_easy_escape* which is presented in Listing 5. If argument `inlength` is provided with a negative value `-1` (`0xFFFFFFFF` in 2's complement), `alloc` is assigned with `0xFFFFFFFF` which is not `0` (Line 3). Without checking the allocation size, the function is forced to allocate an overlarge buffer at Line 6. In the version of 7.50.3, the vulnerability is eliminated by adding the code:

```
if (inlength < 0) return NULL;
```

After successfully locating the defective function *curl_easy_escape* in Curl, BINPATCH extracts the patch code from Curl v7.50.3, and replaces the argument `inlength` with that of the defective function:

```
cmp     [esp+48h], 0
jl      loc_805B4DA ; the function epilogue
```

`0x48` is the offset of `inlength` on the stack of the defective function. BINPATCH then inserts the patch code before Line 3 in Listing 5 from the perspective of source code.

D. THREATS TO VALIDITY

For automatic source code repair (or fixing, patching), there have been sophisticated benchmarks to evaluate the performance of a newly proposed method, such as ManyBugs [36] for C/C++, Defects4J [31] for Java, etc. However, there is no such benchmarks for binary code patching. Due to the inherent complexity of binary code analysis, we are only able to study eight real-world vulnerabilities in details for the evaluation, showing BinPatch is effective in locating and patching them, which cannot cover all cases. The threat could be reduced by employing more real-world instances of vulnerabilities in the future.

VI. DISCUSSION

A. THE SCOPE OF BINPATCH

BinPatch aims to assist analysts or maintainers in locating and patching known vulnerabilities in binary programs. The analysts have the knowledge of the program functionality and whether the program might contain the vulnerability. Since the vulnerability has been known, the error-triggering test case is available as well.

BinPatch is not designed to patch all kinds of vulnerabilities, but those handled by adding new conditions or modifying the original ones (e.g., buffer overflows). It generates patch code basing on the differential path condition between the correct and defective function. It becomes ineffective if the control flows of the two functions are the same that their path conditions are identical. Additionally, on the binary level, members of a data structure are accessed with offsets of constant values. Thus, BinPatch performs patching under the condition that the data structures of the two functions should be the same. Namely, the same member of a data structure should be accessed with identical offset. Otherwise, the patch code would access wrong member variables in the defective function, and produce wrong behaviors.

Obfuscation and code refactoring are out of the scope of this paper. On one hand, techniques and solutions for deobfuscation have been well studied currently, such as bit-level taint analysis [77], VMHunt [74]. If the object binary code is found to be obfuscated, it is better to deobfuscate it firstly, then perform further analysis. On the other hand, BinPatch needs to locate the defective function for the following patching. It cannot handle the cases if the function is removed or combined into others because of refactoring.

In the section of Methodology (§III), we present how BinPatch patches the vulnerability by inserting code at one place of a function. Each time, it captures one patch condition once it finds the first unaligned element of the two path conditions. If a patch modifies multiple places of a function, after inserting the patch code at the first place, BinPatch then re-emulates the defective function, and generates patch code for the next place.

B. FUNCTION INLINING

BinPatch cannot handle inlined functions. On one hand, it relies on CACompare, the binary code similarity comparison technique, to locate the defective function. The accuracy of CACompare thus affects the performance of BinPatch. CACompare cannot locate the inlined functions, which is still an issue for the topic [9], [17], [27], [55]. As a result, BinPatch is unable to patch the binary functions which are inlined into their callers. On the other hand, BinPatch generates patch code basing on the (emulated) execution of the defective function and its corrected version. It requires the same input of the two functions. If the defective function is inlined, it is difficult to decide the input of its caller, which should ensure the corresponding input of the defective function to be the same as that of the correct function.

C. BINARY REWRITING

BinPatch patches code by adding new `.text` sections which are connected with original code through long jumps (① and ② in Figure 7). There are also other methods to rewrite binaries. For instance, inserting code to gaps between functions. But the size of a gap is usually not enough for a patch. Another way is reassembling which injects patch to the original code and relocates it [68], [70]. Since the long jumps added by BinPatch might violate security requirements (e.g., Control Flow Integrity [1]), reassembling is the future choice for BinPatch.

D. PATCH RESTORATION

When analyzers or maintainers find that BinPatch patches a wrong function, BinPatch restores the modified function with the patching log generated after Binary Rewriting (§III-C). Then, if BinPatch is directly provided with the target defective function, it would attempt to patch that function. Otherwise, it would process the next function on the candidate defective function list generated after Defective Function Localization (§III-A).

VII. RELATED WORK

A. AUTOMATIC PROGRAM REPAIR

Automatic program repair (or fixing, patching) is motivated by the high costs of repairing defective programs. Numerous solutions have been proposed for source code repair [16], [23], [30], [33], [34], [37], [40], [42], [45], [46], [52], [63], [64], [72], [73], [76]. As the source code is available, they have *type information* and *symbol names* for program synthesis. Besides, they assume they have plenty of *test cases* to detect defective code and generate the patch. We would not discuss that type of work in more details because this paper focuses on binary code patching where the source code is unavailable and test cases are inadequate. Next, we mainly discuss the solutions for binary code patching.

Table 3 summaries the attributes of different solutions for binary code patching. EC [56]–[58] adopts the technique of SBFL (Spectrum-based Fault Localization) to identify the

TABLE 3. Comparison among different solutions of binary code patching.

Solution	Type	Object Binaries	Fault Localization	Patch Generation
EC [56]–[58]	Static	Stripped	Spectrum-based Fault Localization	Search-based
ClearView [54]	Dynamic	Stripped	Execution Verification	Learning-based
SafeStack [10]	Dynamic	Stripped	Execution Verification	Model-based
KARMA [11]	Static	Non-stripped	String Matching	Example-based
BINPATCH	Static	Stripped	Binary Code Similarity Comparison	Example-based

faulty binary instructions, which still requires numerous test cases to fulfill the target. It also needs test cases to specify the intended behaviors in order to search for the correct patch code (search-based patch generation). ClearView [54] and SafeStack [10] focus on overflow vulnerabilities, including buffer overflows, integer overflows, and heap overflows. They monitor running programs and patch them if they detect erroneous behaviors (Execution Verification), e.g., buffer overflowing. ClearView generates patch code by firstly learning runtime invariants from normal execution of the object binary program (Learning-based). It then alters the control flow when the invariants are violated. SafeStack fulfills patch generation basing on First-Aid [23] which leverages predefined code templates to correct overflow bugs (Model-based). KARMA [11] patches the known vulnerabilities in Android kernels. Commonly, the binary code of Android kernels has the symbol information, i.e., non-stripped. Thus, with bug reports, the corresponding defective code could be located easily basing on symbol names (String Matching). Additionally, because of the nature of Linux (Android) kernels that the kernels are stable across multiple releases and well maintained, KARMA refers to the correct version of kernels, e.g., from upstream source, to extract patch code (Example-based).

Source code program repair is aimed at the processes of software development and testing, while binary program patching is software maintenance for released binary code which is usually stripped. There would be enough test cases for source code program repair for locating faulty code and verifying the correctness of candidate patch code, which is not realistic for binary code. BINPATCH aims to patch the known vulnerabilities in stripped binaries which lack test cases. It locates the defective function via static binary code similarity comparison. It merely adopts a few error-triggering test cases, which are accessible in vulnerability databases, to extract patch code from the correct code (Example-based), but does not depend on them to specify the intended correct behaviors. Thus, comparing to EC which needs a number of test cases and KARMA which is merely designed for non-stripped binaries, BINPATCH is more effective in patching real-world binary programs. Besides, ClearView captures invariants for patch generation from normal executions, and SafeStack generates patch code with pre-defined templates. It is difficult for them to cover all behaviors of the correct patch code. In contrast, BINPATCH extracts the corresponding code from the correct version of the object binary program as the patch which is the best practice to remove the vulnerabilities.

B. BINARY CODE SIMILARITY COMPARISON

The technique of binary code similarity comparison (or clone detection) has been well studied, and has numerous applications in software engineering and security.

Jhi *et al.* [29] and Zhang *et al.* [78] leverage invariants during the execution of binary code to detect software and algorithm plagiarism. Ming *et al.* [48] infer malware lineage via code clone comparison. However, above methods only analyze executed binary code and cannot cover all functions of the target program. Egele *et al.* [18] propose blanket execution to detect similar binary code. They break the normal execution of binary code to pursue high code coverage, while the detection accuracy decreases. Luo *et al.* [44] and Zhang *et al.* [79] adopt symbolic execution to software plagiarism. They rely on the performance of SAT/SMT solvers which cannot handle all cases. David *et al.* [12]–[15] break every basic block of a binary function into program slices. Each slice corresponds to an output of a basic block. The similarity of two functions is measured by counting the identical slices they have.

Multi-MH [55], discovRE [20], and Genius [21] are proposed to detect known bugs in binaries via code similarity analysis. However, discovRE and Genius base on control flow graphs which could be changed significantly because of different compilation configurations (e.g., variant compilers). Therefore, they are ineffective in comparing code compiled with different configurations in this paper. More recently, BinGo [9], CACompare [27], and IMF-sim [71] are proposed to detect similar binary code as well. Besides, BinShape [61] and BinSequence [28] are proposed to measure the similarity of binary function accurately and efficiently. For a reference

function, they adopt filters to remove irrelevant target functions basing on syntax features. Then, BinShape adopts B++ tree to index feature vectors of functions for similarity comparison, while BinSequence explores function paths to infer the similarity of the reference function and each target function in pairs, which is similar to CoP [44]. BinArm [60] aims to detect vulnerabilities in firmware images. Similar to BinShape and BinSequence, it is also a multi-stage solution which adopts the filtering process to ensure the efficiency, then compares the similarity of functions via fuzzy matching of their control flow graphs. FOSSIL [2] is proposed to identify free open-source software functions in malwares. It adopts opcodes, paths of control flow graphs and opcode frequency distribution as code features. Then it applies z-score to the features and leverages the Bayesian network to calculate the similarity score of two functions.

Machine learning techniques are also applied to the topic. Xu *et al.* [75] train the neural network with the feature vectors extracted from control flow graphs of binary functions. Liu *et al.* [39] directly leverage machine code to compute best parameters for their neural network model. Ding *et al.* [17] treat a binary function as a document, and handle the problem with NLP (Natural Language Processing) techniques.

BINPATCH is proposed to patch known vulnerabilities in binary programs. It is worth nothing that the process of fault localization is not to detect new bugs, but to locate the known bugs in object binaries. Thus, in this paper, CACompare, an existing similar binary code detector, is adopted to fulfill the process, which is substitutable if there exists a better solution catering to the scenarios and requirements.

C. BINARY REWRITING

Binary rewriting is the process that transforms one binary into another, adding new metrics, such as security, while maintains the original functionality [65]. The rewriting could be realized either statically or dynamically. The static solutions perform offline modifications to the binaries, introducing lower overhead, while the dynamic solutions instrument the binary code at runtime, which guarantee a full-coverage transformation of COTS (Commercial Off-The-Shelf) or stripped binaries [68].

1) STATIC BINARY REWRITING

Early static solutions focus on object files which contain debug and symbol information. With the information, it is easy to differentiate code from data and decide the targets of indirect calls/jumps. SASI [19] adds extra code to the object file for the security basing on the information provided by the compiler. In the similar way, Plot [59] and Diablo [67] fulfill code optimization and profiling via rewriting. BIRD [49] and PSI [81] are typical solutions which perform rewriting on COTS via detouring. They insert jump-out code, adding new text sections, to add new code. SecondWrite [53] is the first solution that implements binary rewriting with IR (Intermediate Representation). It lifts binary code into LLVM-IR, then translates the IR back to machine code after adding the new code. Zipr [24] and Zipr++ [25] are designed in the similar way to enforce the security of the original code. Dyninst [6] disassembles the binary function and extracts its control flow graph, then inserts new basic blocks into the graph. Uroboros [69], [70] leverages the technique of reassembling which re-computes the addresses and offsets after inserting new code. It avoids the huge overhead introduced by detours (i.e. long jumps) or full IR translation. Ramblr [68] further eliminate several assumptions of previous work, making reassembling more robust for real-world cases. Multiverse [5] abandons the heuristics adopted by other solutions. It constructs mappings to find the corresponding new addresses of the original code and data. In addition to the above solutions working on the Intel architecture, Kim et al. propose RevARM [35] to rewrite binary code of ARM. They instrument the internal representations transformed from the original binaries, then generate the new binary code.

2) DYNAMIC BINARY REWRITING

Dynamic binary instrumentation (DBI) is the typical application of dynamic binary rewriting. With the runtime information, it is easier to obtain the targets of indirect calls and differentiate pointers from constants according to their usage (e.g., memory accessing, logic arithmetic). Currently, dynamic rewriting tools DynamoRIO [7], [8], Valgrind [50], [51] and Pin [43] are widely used for DBI. The predecessor of DynamoRIO, Dynamo [3], is a dynamic optimization system for code execution. Basing on Dynamo, DynamoRIO further provide various lightweight APIs (Application Programming Interfaces) which could monitor the execution of any binary instruction. Valgrind is a heavyweight solution for binary instrumentation. In addition to the APIs, it also introduces profiling tools, such as Memcheck, to analyze the runtime memory of the object program. Pin implements a virtual machine which takes the original code and instrumentation code as input, then handles the new code with a JIT (just-in-time) compiler and emulates the execution.

The focus of binary rewriting is how to insert new code to the original binaries, while that of binary patching is how to locate the defective code and generate patch code (the new code) to correct it. Binary patching needs the technology of binary rewriting to insert the patch code. In this paper, BINPATCH implements the rewriting with detouring which could be substituted with reassembling in the future (§VI-C).

VIII. CONCLUSION

In this paper, we present BINPATCH to automatically patch known vulnerabilities of binary programs. It locates defective code via similar code comparison, and generates patch code by reusing corresponding code from the correct version of the defective code. BINPATCH is evaluated with eight real-world vulnerabilities. The experimental results indicate that it is able to not only locate the defective functions in binary

programs, but also generate the correct patch code so as to remove the vulnerabilities.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments which greatly help to improve the manuscript. They would like to acknowledge the support from the Ant Financial Services Group as well.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Comput. Commun. Secur.*, Nov. 2005, pp. 340–353.

[2] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "*FOSSIL*: A resilient and efficient system for identifying foss functions in malware binaries," *ACM Trans. Privacy Secur.*, vol. 21, no. 2, Feb. 2018, Art. no. 8.

[3] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2000, pp. 1–12.

[4] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Proc. Int. Conf. Compiler Construction*. Berlin, Germany: Springer, 2004, pp. 5–23.

[5] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proc. 25th Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2018, pp. 40–47.

[6] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proc. 10th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools*, Sep. 2011, pp. 9–16.

[7] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2003, pp. 265–275.

[8] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 133–144, Jul. 2012.

[9] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-OS binary search," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Nov. 2016, pp. 678–689.

[10] G. Chen *et al.*, "Safestack: Automatically patching stack-based buffer overflow vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 10, no. 6, pp. 368–379, Nov./Dec. 2013.

[11] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *Proc. 26th USENIX Secur. Symp. (USENIX Security)*, 2017, pp. 1253–1270.

[12] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2016, pp. 266–280.

[13] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2017, pp. 79–94.

[14] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2018, pp. 392–404.

[15] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2014, pp. 349–360.

[16] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard, "Inference and enforcement of data structure consistency specifications," in *Proc. 2006 Int. Symp. Softw. Test. Anal.*, Jul. 2006, pp. 233–244.

[17] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2019, pp. 38–55.

[18] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proc. 23rd USENIX Secur. Symp. (USENIX Security)*, Aug. 2014, pp. 303–317.

[19] U. Erlingsson and F. B. Schneider, "Sasi enforcement of security policies: A retrospective," in *Proc. DARPA Inf. Survivability Conf. Expo.. DISCEX*, vol. 2, Jan. 2000, pp. 287–295.

[20] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "DiscovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, 2016.

[21] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Oct. 2016, pp. 480–491.

[22] A. Fog, "Calling conventions for different C++ compilers and operating systems," Tech. Univ. Denmark, Copenhagen, Denmark, Tech. Rep., 2018.

[23] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing Q&A sites (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 307–318.

[24] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson, "Zipr: Efficient static binary rewriting for security," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 559–566.

[25] J. Hiser, A. Nguyen-Tuong, W. Hawkins, M. McGill, M. Co, and J. Davidson, "Zipr++: Exceptional binary rewriting," in *Proc. Workshop Forming Ecosyst. Around Softw. Transformation*, Nov. 2017, pp. 9–15.

[26] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, vol. 1, Mar. 2016, pp. 57–67.

[27] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proc. 25th Int. Conf. Program Comprehension (ICPC)*, May 2017, pp. 88–98.

[28] H. Huang, A. M. Youssef, and M. Debbabi, "BinSequence: Fast, accurate and scalable binary code reuse detection," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 155–166.

[29] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, May 2011, pp. 756–765.

[30] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 389–400, Jun. 2011.

[31] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2014, pp. 437–440.

[32] U. Kargén and N. Shahmehri, "Turning programs against each other: High coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, Aug./Sep. 2015, pp. 782–792.

[33] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 295–306.

[34] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 802–811.

[35] T. Kim *et al.*, "Revarm: A platform-agnostic ARM binary rewriter for security applications," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Dec. 2017, pp. 412–424.

[36] C. Le Goues *et al.*, "The manybugs and introclass benchmarks for automated repair of C programs," *IEEE Trans. Softw. Eng.*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015.

[37] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.

[38] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, pp. 2201–2215, Oct./Nov. 2017.

[39] B. Liu *et al.*, "αDIFF: Cross-version binary code similarity detection with DNN," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, New York, NY, USA, Sep. 2018, pp. 667–678.

[40] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2fix: Automatically generating bug fixes from bug reports," in *Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation*, Mar. 2013, pp. 282–291.

[41] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proc. 38th Int. Conf. Softw. Eng.*, pp. 702–713, May 2016.

[42] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 298–312, Jan. 2016.

[43] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, Jun. 2005.

[44] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 389–400.

[45] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, pp. 129–139, May/Jun. 2018.

[46] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 691–701.

[47] X. Meng and B. P. Miller, "Binary code is not easy," in *Proc. 25th Int. Symp. Softw. Test. Anal. (ISSTA)*, pp. 24–35, Jul. 2016.

[48] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 253–270.

[49] S. Nanda, W. Li, L.-C. Lam, and T.-C. Chiueh, "Bird: Binary interpretation using runtime disassembly," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2006, pp. 358–370.

[50] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 2, pp. 44–66, Oct. 2003.

[51] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

[52] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, pp. 772–781, May 2013.

[53] P. O'sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting security in cots software with binary rewriting," in *Proc. IFIP Int. Inf. Secur. Conf.* Berlin, Germany: Springer, 2011, pp. 154–172.

[54] J. H. Perkins *et al.*, "Automatically patching errors in deployed software," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, pp. 87–102, Oct. 2009.

[55] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 709–724.

[56] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated repair of binary and assembly programs for cooperating embedded devices," in *Proc. 18th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, New York, NY, USA, pp. 317–328, Mar. 2013.

[57] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, New York, NY, USA, Sep. 2010, pp. 313–316.

[58] E. M. Schulte, W. Weimer, and S. Forrest, "Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair," in *Proc. Companion Publication Annu. Conf. Genetic Evol. Comput. (GECCO)*, New York, NY, USA, Jul. 2015, pp. 847–854.

[59] B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "PLTO: A link-time optimizer for the intel IA-32 architecture," in *Proc. Workshop Binary Transl. (WBT)*, Feb. 2001.

[60] P. Shirani *et al.*, "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2018, pp. 114–138.

[61] P. Shirani, L. Wang, and M. Debbabi, "BinShape: Scalable and robust binary library function identification using function shape," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment* Berlin, Germany: Springer, Jun. 2017, pp. 301–324.

[62] Y. Shoshitaishvili *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 138–157.

[63] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "Codecarboncopy," in *Proc. 11th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, Sep. 2017, pp. 95–105.

[64] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 43–54, Jun. 2015.

[65] M. Smithson, K. ElWazeer, K. Anand, A. Kotha, and R. Barua, "Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness," in *Proc. 20th Working Conf. Reverse Eng. (WCRE)*, Oct. 2013, pp. 52–61.

[66] J. Sohn and S. Yoo, "FLUCCS: Using code and change metrics to improve fault localization," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, pp. 273–283, Jul. 2017.

[67] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "DIABLO: A reliable, retargetable and extensible link-time rewriting framework," in *Proc. 5th IEEE Int.Symp. Signal Process. Inf. Technol.*, Dec. 2005, pp. 7–12.

[68] R. Wang *et al.*, "Ramblr: Making reassembly great again," in *Proc. 24th Annu. Symp. Netw. Distrib. Syst. Secur. (NDSS)*, Feb. 2017.

[69] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proc. USENIX Secur. Symp.*, Aug. 2015, pp. 627–642.

[70] S. Wang, P. Wang, and D. Wu, "UROBOROS: Instrumenting stripped binaries with static reassembling," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, vol. 1, Mar. 2016, pp. 236–247.

[71] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Oct./Nov. 2017, pp. 319–330.

[72] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. 31st Int. Conf. Softw. Eng.*, pp. 364–374, May 2009.

[73] Y. Xiong *et al.*, "Precise condition synthesis for program repair," in *Proc. 39th Int. Conf. Softw. Eng.*, May 2017, pp. 416–426.

[74] D. Xu, J. Ming, Y. Fu, and D. Wu, "VMHunt: A verifiable approach to partially-virtualized binary code simplification," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, pp. 442–458, Oct. 2018.

[75] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, Oct./Nov. 2017, pp. 363–376.

[76] J. Xuan *et al.*, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017.

[77] B. Yadegari and S. Debray, "Bit-level taint analysis," in *Proc. IEEE 14th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2014, pp. 255–264.

[78] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, Jul. 2012, pp. 111–121.

[79] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2014, pp. 66–77.

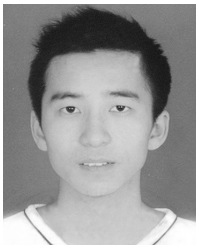
[80] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *Proc. 27th USENIX Secur. Symp.*, Baltimore, MD, USA, Aug. 2018, pp. 887–902.

[81] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," *ACM SIGPLAN Notices*, vol. 49, no. 7, pp. 129–140, Jul. 2014.

[82] X. Zhang and R. Gupta, "Cost effective dynamic program slicing," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2004, pp. 94–106.

[83] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Proc. 25th Int. Conf. Softw. Eng. (ICSE)*, pp. 319–329, May 2003.

[84] Z. Zhang and X. Koutsoukos, "Generic value-set analysis on low-level code," in *Proc. 5th Analytic Virtual Integr. Cyber-Phys. Syst. Workshop*. Rome, Italy: Linköping Univ. Electron. Press, Dec. 2014.



YIKUN HU received the B.S. degree in computer science and engineering from the South China University of Technology, Guangzhou, China. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His main research interests include binary program analysis and software repair.

YUANYUAN ZHANG received the B.S. and M.S. degrees from Wuhan University and the Ph.D. degree from Shanghai Jiao Tong University (SJTU). Before joining SJTU, she was a Postdoctoral Fellow with INSA de Lyon. She is currently an Associate Professor with the Department of Computer Science and Engineering, SJTU. Her research interests include network security, architecture security, and system security.

DAWU GU received the B.S. degree in applied mathematics from the Xidian University of China, in 1992, and the M.S. and Ph.D. degrees in cryptography from the Xidian University of China, in 1998. He is currently a Distinguished Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University (SJTU), China. He leads the Laboratory of Cryptology and Computer Security, SJTU. He has published over 150 scientific papers in academic journals and at conferences. He holds 30 innovation patents. His research interests include cryptography, side channel attack, and software security. He serves as an Executive Member of the China Association of Cryptologic Research and a member of the China Computer Federation. He was the winner of the Chang Jiang Scholars Program by the Ministry of Education of China. He was a recipient of the Prize of the National Scientific and Technological Progress of China, in 2017. He has been invited as a Chair and a TPC Member for many conferences and workshops. He also serves as several technical editors for *China Communications*, the *Journal of Cryptologic Research*, *Information Network Security*, and so on.