2017 IEEE/ACM 39th International Conference on Software Engineering

# SPAIN: Security Patch Analysis for Binaries Towards Understanding the Pain and Pills

Zhengzi Xu*, Bihuan Chen*‡, Mahinthan Chandramohan*, Yang Liu* and Fu Song†
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
†School of Information Science and Technology, ShanghaiTech University, China
‡Corresponding Author

***Abstract*—Software vulnerability is one of the major threats to software security. Once discovered, vulnerabilities are often fixed by applying security patches. In that sense, security patches carry valuable information about vulnerabilities, which could be used to discover, understand and fix (similar) vulnerabilities. However, most existing patch analysis approaches work at the source code level, while binary-level patch analysis often heavily relies on a lot of human efforts and expertise. Even worse, some vulnerabilities may be secretly patched without applying CVE numbers, or only the patched binary programs are available while the patches are not publicly released. These practices greatly hinder patch analysis and vulnerability analysis.**

**In this paper, we propose a scalable binary-level patch analysis framework, named SPAIN, which can automatically identify security patches and summarize patch patterns and their corresponding vulnerability patterns. Specifically, given the original and patched versions of a binary program, we locate the patched functions and identify the changed traces (i.e., a sequence of basic blocks) that may contain security or non-security patches. Then we identify security patches through a semantic analysis of these traces and summarize the patterns through a taint analysis on the patched functions. The summarized patterns can be used to search similar patches or vulnerabilities in binary programs.**

**Our experimental results on several real-world projects have shown that: i) SPAIN identified security patches with high accuracy and high scalability; ii) SPAIN summarized 5 patch patterns and their corresponding vulnerability patterns for 5 vulnerability types; and iii) SPAIN discovered security patches that were not documented, and discovered 3 zero-day vulnerabilities.**

## I. Introduction

Program vulnerability is one of the major threats to software security. However, it is almost impossible to avoid vulnerabilities at the development stage; and it is even difficult to discover vulnerabilities at the production stage. Security experts usually leverage dynamic fuzzing (e.g., [1, 2]), symbolic execution (e.g., [3, 4]) or static code auditing (e.g., [5, 6]) to find vulnerabilities. However, none of these techniques can provide a complete solution to win the war against vulnerabilities. Dynamic fuzzing suffers from the code coverage problem and the initial seeds problem [7]. Symbolic execution cannot scale well to real-world programs, due to path explosion and constraint solving problems. Static code auditing often requires human expertise, and cannot scale well when the program complexity increases.

Vulnerabilities, once discovered, are often fixed by applying security patches. In that sense, security patches carry important information about vulnerabilities. By focusing on the remedy of vulnerabilities instead of the vulnerabilities themselves, patch analysis has been proposed to discover n-day vulnerabilities, whose patches have been released but not deployed to every instance of the software in the world. Since patch analysis is relatively accurate to find vulnerabilities, it has gained the popularity in both industry and academia. Also, security patches are good entry points to understanding the program weaknesses and how the vulnerabilities work inside the program, especially for security participants who do not have access to the source code. Moreover, the underlying information of patches has been used to build automatic bug-fixing tools [8, 9, 10], generating valid patches for similar vulnerabilities. However, up till today, most existing researches on patch analysis work at the source code level [11, 12], but very few works have been done to tackle this problem at the binary-level.

Binary-level patch analysis can only be performed on machine instructions for closed source programs without symbol tables, which often requires a significant amount of human efforts and expertise to understand the semantics of instructions. Due to this complex nature, the existing techniques [13, 14] often heavily rely on manual or heavy program analysis, which becomes infeasible for real-world programs.

On the other hand, software companies may tend to patch the vulnerabilities they find themselves in a secret way instead of making them public and applying Common Vulnerabilities and Exposures (CVE) numbers due to their security regulations or policies. As a result, security analysts cannot know the existence of particular vulnerabilities, which hinders the understanding and analysis of vulnerabilities. Even worse, in some cases, only the patched binary programs are available; while the patches themselves may not be publicly released, which hinders the existing patch analysis techniques that often rely on the availability of patches. Moreover, due to the application of patch obfuscation and patch modification techniques such as honey-patching [15], the patch patterns in open source binaries may differ greatly from close source ones.

To address these problems, we propose a scalable binary-level patch analysis framework, named SPAIN, to automatically identify security patches and summarize patch patterns and their corresponding vulnerability patterns. In particular, given the original and patched versions of a binary program, SPAIN locates the functions that have been changed from the original binary to the patched binary. Then, it detects the changed traces (i.e., a sequence of basic blocks) for each patched function to capture the function-level changes. These traces may contain security or non-security patches. Finally, it identifies security patches through a semantic analysis of these traces and summarizes the patterns through a taint analysis on the patched functions. As we run more binary programs, we can maintain a database of patterns that will be continuously enriched to embrace the evolving and emerging of various vulnerabilities. There are many impactful applications of the learned patterns in binary security like patch detection, vulnerability detection, automatic patch synthesis, and even patch/vulnerability trend analysis with the help of data analytic techniques. A more detailed discussion can be found in Section III-E.

We evaluated SPAIN on several real-world projects. The experimental results demonstrated that SPAIN can identify security patches with high accuracy and high scalability, and summarize 5 security patch patterns and their corresponding vulnerability patterns for 5 types of vulnerabilities. Moreover, we discovered undocumented security patches and found 3 zero-day vulnerabilities in Adobe PDF Reader.

In summary, our work makes the following contributions.

- We proposed a scalable binary-level patch analysis framework SPAIN, which can identify security patches and summarize patch patterns and their corresponding vulnerability patterns.
- We implemented SPAIN in a prototype and conducted experiments to demonstrate the accuracy, scalability, and application of SPAIN.
- We discovered undocumented security patches and found 3 zero-day vulnerabilities in Adobe PDF Reader.

## II. Preliminaries and Overview

### A. Preliminaries

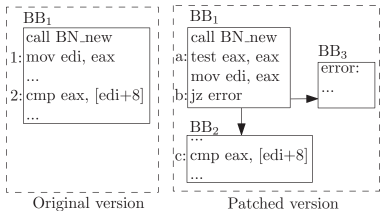
In this work, we assume that binary programs are in the X86 32bit format. Each binary program contains a number of functions with an entry point (starting function). This section introduces several basic concepts in binaries, i.e., basic block, control flow graph and partial trace.

**Definition 1:** A *basic block* in a binary function is a straight-line sequence of X86 instructions with no branches in except to the entry and no branches out except at the exit.

**Definition 2:** Given a binary function, the *control flow graph* (CFG) of the function is a tuple $G = (N,\ E,\ N_s,\ N_t,\ \iota)$, where $N$ is a finite set of nodes, and each node represents a basic block in the function, $E : N \times N$ is a set of edges that connect two nodes and represent the control flow from one basic block to the another, $N_s,\ N_t \subset N$ are respectively the sets of start and end points of the function, and $\iota$ is a function associating every node $n \in N$ with the basic block $\iota(n)$.

Notice that there might be more than one start point for a function, because it is hard to precisely identify the boundary of a function at the binary level [16].

**Definition 3:** Given a CFG $G = (N,\ E,\ N_s,\ N_t,\ \iota)$ of a function, a *partial trace* $t$ in $G$ is a finite sequence of nodes $\langle n_1,\ n_2,\ \ldots,\ n_k\rangle$ for some $k \geq 1$, where $(n_i,\ n_{i+1}) \in E$, for every $i:\ 1 \leq i < k$.



```
   call BN_new
1: mov edi, eax
   ...
2: cmp eax, [edi+8]
   ...
```

```
   call BN_new
a: test eax, eax
   mov edi, eax
b: jz error

error:
...

...
c: cmp eax, [edi+8]
...
```

Fig. 1: Running Example: NULL Pointer Dereference Vulnerability and its Patch from OpenSSL 1.0.1l.

### B. Running Example

Figure 1 lists the partial X86 assembly code of one `NULL` pointer dereference vulnerability abstracted from the OpenSSL 1.0.1l that consists of the original version and patched version. In the original version, the program first calls the function `NB_new` by which the return value is stored in the register `eax`. Then, it assigns the return value to the register `edi` at the location 1. Later, the return value is directly used to dereference the memory `[edi+8]` at the location 2. This vulnerability is patched in OpenSSL 1.0.1m by checking whether the return value of the function call `NB_new` is `NULL` or not. This checking is implemented in the patched version by adding two instructions: `test eax, eax` and `jz error`. It first tests the return value at the location $a$ by `test eax, eax` which performs a bitwise `AND` operation on the operand `eax`. The test operation sets the carry flag `cf` and overflow flag `of` to 0. The sign flag `sf` is set to the most significant bit of the result of the bitwise `AND` operation. The zero flag `zf` is set to 1 if the result of `AND` operation is 0, 0 otherwise. The parity flag `pf` is set to 1 if the number of ones in that byte is even, 0 otherwise. The value of the adjust flag `af` is undefined. After assigning the return value to the register `edi`, it checks whether the zero flag `zf` is 1 or not (i.e., the value of `eax` is `Null` or not) at the location $b$. If it is 1 (i.e., `eax` is `NULL`), the control flow will jump to the location `error` for error handling. Otherwise, the return value is used to dereference the memory `[edi+8]` at the location $c$.

In this running example, there are one basic block $\texttt{BB}_1$ and three basic blocks $\texttt{BB}_1$, $\texttt{BB}_2$ and $\texttt{BB}_3$ respectively in the original version and patched version. The CFG of the original version is $G_o = (\{n_1\},\ \emptyset,\ \{n_1\},\ \{n_1\},\ \iota)$, where $\iota(n_i)$ $=\texttt{BB}_1$. The CFG of the patched version is $G_p = (\{n_1, n_2, n_3\},\ \ \{(n_1, n_2), (n_1, n_3)\},\ \ \{n_1\},\ \ \{n_2, n_3\},\ \ \iota)$, where $\iota(n_i)$ $=\texttt{BB}_i$ for every $i:\ \ 1 \leq i \leq 3$. There are two partial traces $\langle n_1,\ n_2\rangle$ and $\langle n_1,\ n_3\rangle$ in $G_p$, which corresponds to two possible execution traces of the patched program. While there is only one partial trace $\langle n_1\rangle$ in $G_o$.

### C. Framework Overview

Figure 2 presents the overview of SPAIN, which consists of four components. Taking the original and patched versions of a binary program as inputs, SPAIN first locates the functions that have been changed from the original version to the patched version and detects the changed traces in each patched function to capture the function-level changes. Then SPAIN determines whether such changes are caused by security or non-security

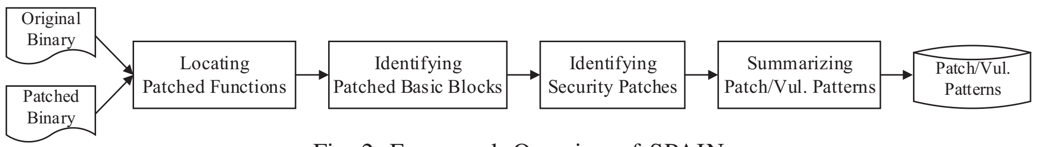

Original Binary
Patched Binary
Locating Patched Functions
Identifying Patched Basic Blocks
Identifying Security Patches
Summarizing Patch/Vul. Patterns
Patch/Vul. Patterns

Fig. 2: Framework Overview of SPAIN

patches through semantic analysis and then summarizes the patterns of security patches and their corresponding vulnerabilities through taint analysis. As we run more binary programs, SPAIN has the capability to continuously learn and accumulate the knowledge of security patches to embrace the evolving and emerging of various vulnerabilities.

**Locating Patched Functions.** Given the original and patched versions of a binary program, this component first uses the disassembler tool IDA Pro [17] and the binary comparison tool BinDiff [18] to obtain matched function pairs in the original and patched versions. Such pairs are called *candidate pairs* that may contain security or non-security patches. To reduce the size of pairs for further analysis and hence improve the scalability, this component removes the pairs from the candidate pairs, in which the functions have no changes or only compiler-introduced changes.

**Identifying Patched Basic Blocks.** For each function pair in the candidate pairs (i.e., the patched and the original functions), this component first leverages the pairwise basic block matching to identify the patched basic blocks within the patched function, identifies the relationships among these patched basic blocks in terms of patched partial traces (see Definition 3) that capture the locality of a patch, and finally determines the original partial traces in the original function that are relevant to each patched partial trace.

**Identifying Security Patches.** For a patched partial trace and its corresponding original partial traces, this component decides whether the changes are caused by a security or non-security patch. It is realized by a semantic analysis to compute the semantic difference between the patched partial trace and each of the original partial trace. We use small semantic difference as the indicator for security patches based on the heuristic that security patches are less likely to introduce new semantics than non-security patches (e.g., feature upgrades).

**Example 1:** Recalling the running example in Figure 1, the difference between the original and patched versions is the sanity check, namely `test eax, eax` and `jz error` in the patched version. The semantic difference between the patched and original versions is very minimal (e.g. $< 0.2$ in our experiments) and hence, we can safely conclude that it is a security patch (see Section III-C for details).

**Summarizing Patch and Vulnerability Patterns.** Once a security patch is identified, this component summarizes the patch pattern and the corresponding vulnerability pattern through a taint analysis from security-sensitive instructions in the patched function. The patterns capture the security-critical sources, sinks and sanity checks. One potential application of the summarized patterns is to search for similar patches or vulnerabilities in binary programs.

**Example 2:** Consider the running example, the function `NB_new` is an external unknown function. The return value of `NB_new` is therefore regarded as the taint source/input. This taint source is directly used to perform the security-sensitive operation, memory dereferencing `[edi+8]`, which is regarded as the taint sink. There is no checking of the tainted source between the source and sink points. From this vulnerability, the vulnerability pattern summarized by SPAIN is shown in the 4th row (NULL Pointer Dereference) of Table IV. Intuitively, this pattern specifies that there is an untrusted function call whose return value is directly used to dereference memory without any sanity checking on them.

While in the patched version of the running example, the return value from the external unknown function `NB_new` is checked for `NULL` value. If it is `NULL`, the program jumps to an error handler. Otherwise, the return value is used to perform the security-sensitive operation, that is memory dereferencing. The patch pattern learned by SPAIN is shown in the 4th row (NULL Pointer Dereference) of Table IV. Intuitively, this patch pattern expresses that before using the return value of an untrusted function call to dereference memory, there must be a sanity checking of the return value.

### D. Assumptions

SPAIN has a few underlying assumptions, which may limit its application and threat its validity. First, we focus on patches in which only one function is modified for one patch, but do not support patches where multiple functions are changed for one patch. Second, we assume that the function matching results generated by IDA Pro [17] and BinDiff [18] are correct, although no tools can reach 100% accuracy [19]. Third, we cannot identify every type of patches since some patches, especially for logical vulnerability patches, may behave like the normal code. Currently, we cover the patches for most common vulnerabilities such as buffer overflows, integer overflows, and double-free/use-after-free. Fourth, we focus on the X86 32bit binary format. We will discuss these assumptions in Section V.

## III. Methodology

### A. Locating Patched Functions

SPAIN starts with locating the matched pairs of functions $\mathcal{F} = \{\langle f_p, f_o\rangle \mid f \text{ is a function in a binary program}\}$, where $f_o$ (i.e., the original function) is changed to $f_p$ (i.e., the patched function) during the patching process. $\mathcal{F}$ is a set of candidate pairs that may contain security or non-security patches.

*1) Function Matching:* In the first step, we leverage the disassembler tool IDA Pro [17] and the binary comparison tool BinDiff [18] to match functions in the original and patched versions of a binary program. In particular, we use IDA Pro to extract the assembly instructions and construct the CFG for each function.

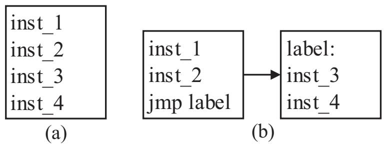


Fig. 3: An Example of Compiler-Introduced Changes

Then, for open source binaries (i.e., with the symbol table), we directly use function names to perform the matching. While for closed source binaries (i.e., without the symbol table), we leverage BinDiff's function matching functionality. Finally, we get the initial candidate pairs, where each function is represented by its CFG. Note that this step is not one of our contributions, and we briefly introduce it to make our methodology complete.

*2) Function Filtering:* Patches usually change a small part of the whole binary program and many functions remain the same. Therefore, in the second step, we remove the function pairs with no changes from the candidate pairs to reduce the size of pairs for further analysis and thus improve the scalability. To this end, we apply the 3D-CFG-based hashing technique [20] to compute the hash value for $f_p$ and $f_o$, and then remove $\langle f_p, f_o \rangle$ from the candidate pairs $\mathcal{F}$ if $f_p$ and $f_o$ have the same hash value.

Further, some changes are introduced by compilers due to the compilation context or optimization level [21]. One common case is that the compiler may split one basic block into two basic blocks, as illustrated in Figure 3. The basic block in Figure 3(a) becomes two basic blocks connected by a `jmp` instruction in Figure 3(b) in the second run of a compiler. However, these two versions are semantically equivalent to each other. Therefore, for any basic block, which has only one successor and its successor has only one predecessor, we merge it with its successor and remove the additional `jmp` instruction.

Another common case is that the compiler might change the operands of the instruction in different runs. For example, the memory addresses are associated with the function position in the binary program and they will change if the function position is changed, which always happens in different runs of a compiler. To account for such changes introduced by compilers, we normalize each basic block, as shown in Algorithm 1. It takes a basic block as an input and returns the normalized basic block. It iteratively normalizes each assembly instruction in the basic block (Line 2). An assembly instruction consists of a mnemonic and up to 3 operands, where a mnemonic represents the specific operation that an instruction performs while an operand is a variable-length sequence of elements (cf. IA-32 [22]). Thus, it gets the mnemonic (Line 3) and normalizes the operands of each instruction according to their operand types (Lines 6–9). An operand can be a register, an immediate value, or a memory, which is respectively normalized to its type `reg`, `imm`, or `mem`. For example, `mov eax, 0x40` is normalized to `mov reg, imm`.

After the merging and normalization of basic blocks for each functions in $\mathcal{F}$, we use the same hashing technique to further remove the pairs that only have some compiler-introduced changes. Our filtering step is designed to be conservative such that security changes are kept.

**Algorithm 1:** Normalize A Basic Block

```
Input  : basic block $b$
Output : normalized basic block $b'$
1  $b' := \langle\rangle$ // sequence of normalized inst.
2  foreach instruction $i$ in $b$ do
3      $m :=$ GetMnemonic($i$)
4      if $m \neq$ 'nop' then
5          $op' := \langle\rangle$ // sequence of operand type
6          foreach operand $o$ in $operands(i)$ do
7              $t :=$ GetOperandType($o$)
               // $t \in \{mem, reg, imm\}$
8              $op' := op' \oplus t$
9          end
10         $i' := \langle m, op' \rangle$ // normalized inst.
11         $b' := b' \oplus i'$
12     end
13 end
14 return $b'$
```

## *B. Identifying Patched Basic Blocks*

Both security and non-security patches can lead to the modifications of various basic blocks and, in the worst case, such modified basic blocks can lie scattered all over a function. Thus, SPAIN proceeds to investigate the candidate pairs $\mathcal{F}$ to identify the basic blocks that are modified (i.e., patched basic blocks) in the patch, identify the relationships among these patched basic blocks in terms of partial traces (see Definition 3), and identify their relations to the original function. Algorithm 2 gives this procedure, which computes the matched trace pairs $\mathcal{T} = \{\langle t_p, \{t_o^1, ..., t_o^n\}\rangle \mid t \text{ is a partial trace in a function}\}$ for each function pair $\langle f_p, f_o \rangle$ in $\mathcal{F}$, where $t_p$ in $f_p$ might be relevant to $\{t_o^1, ..., t_o^n\}$ in $f_o$.

In detail, we leverage the pairwise basic block matching to identify the patched basic blocks within the patched function (Lines 2–13). In particular, for each basic block $b_p$ in the patched function $f_p$, we search for an equivalent basic block $b_o$ in the original function $f_o$. If there is no such a matching basic block $b_o$ in $f_o$, $b_p$ is identified as a patched basic block.

Once the patched basic blocks are identified, we proceed to determine the relationships among these basic blocks (Line 14), i.e., to connect the patched basic blocks to infer the effect of a patch. To this end, for each patched basic block, we leverage the predecessor/successor information to connect related patched blocks; i.e., $b_p^1$ and $b_p^2$ are connected when there is a predecessor/successor relationship between them. In such a manner, patched partial traces (i.e., $\{t_p\}$) are constructed.

To get a clear understanding of how a patched partial trace $t_p$ is related to the original function, we extract the unmodified first-degree neighbors of the patched partial trace (Line 16). Specifically, for each basic block in the patched partial trace, we identify the first-degree neighboring blocks that are unmodified from the original function. These unmodified neighboring basic blocks capture the locality of the patch and help to locate the corresponding basic blocks in the original function (Line 17). In this fashion, for each patched partial trace, we identify the corresponding basic blocks of interest in the original function, and construct the original partial traces $\{t_o^1, ..., t_o^n\}$ in the same

**Algorithm 2:** Identify Partial Traces

```
   Input   : patched function $f_p$, original function $f_o$
   Output : set of matched partial trace pairs $\mathcal{T}$
1  $\mathcal{T} := \emptyset$
2  $\mathcal{B}_p := \emptyset$ // set of patched blocks
3  foreach basic block $b_p$ in $f_p$ do
4      foreach basic block $b_o$ in $f_o$ do
5          if $b_p == b_o$ then
6              $FoundMatch = True$
7              break
8          end
9      end
10     if not $FoundMatch$ then
11         $\mathcal{B}_p := \mathcal{B}_p \cup \{b_p\}$
12     end
13 end
14 $\mathcal{T}_p$:= LinearConnectedComponents($\mathcal{B}_p$)
15 foreach patched partial trace $t_p$ in $\mathcal{T}_p$ do
16     $neighbors$:= GetFirstDegreeNeighbors($t_p$, $f_p$)
17     $\mathcal{B}_o$:= GetRelevantOriginalBlocks($neighbors$, $f_o$)
18     $\mathcal{T}_o$:= LinearConnectedComponents($\mathcal{B}_o$)
19     $\mathcal{T} := \mathcal{T} \cup \{\langle t_p, \mathcal{T}_o\rangle\}$
20 end
21 return $\mathcal{T}$
```

way to construct patched partial traces (Line 18).

**Example 3:** Consider the original and patched functions in Figure 4, where each letter represents a basic block. We can see that in the patched function, the basic block $b$ in the original function is modified to $b'$ and a new basic block $g'$ is added. From the patched function, only one patched partial trace can be extracted, i.e., $\langle b', g' \rangle$, where its unmodified first-degree neighbors are $\{a, c, d\}$. By looking at the first-degree neighbors of the patched partial trace, we can infer that only the basic block $b$ is the corresponding basic block of interest in the original function, and only one original partial trace can be constructed, i.e., $\langle b \rangle$. Hence, in this case, the patched partial trace $\langle b', g' \rangle$ is only related to one original partial trace $\langle b \rangle$.

### C. Identifying Security Patches

Once the patched partial traces and their related partial traces in the original function are identified, we proceed to determine for each pair $\langle t_p, \{t_o^1, ..., t_o^n\}\rangle$ in $\mathcal{T}$ whether the changes are caused by a security patch or non-security patch. To this end, we perform a semantic analysis on both the patched partial trace $t_p$ and the set of original partial traces $\{t_o^1, ..., t_o^n\}$.

The idea underlying our semantic analysis is that *"a security patch is less likely to change the semantics of the underlying function, while a non-security patch is more likely to introduce new semantics"*. Therefore, we compare the *semantic summaries* generated for the patched partial trace and each of the original partial traces by Equation 1.

$$\delta_d = \mathcal{S}_p - \mathcal{S}_o \qquad (1)$$

$\delta_d$ is the semantic difference between the semantic summary $\mathcal{S}_p$ of the patched partial trace $t_p$ and the semantic summary $\mathcal{S}_o$ of the original partial trace $t_o$ in $\{t_o^1, ..., t_o^n\}$. If there exists one original partial trace such that the semantic difference is

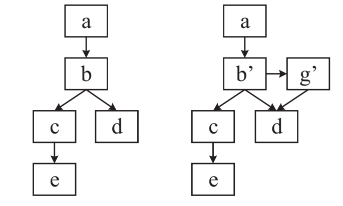


Fig. 4: An Example of the Original and Patched Functions

small, we mark the patch as a security patch. Otherwise, we mark the patch as a non-security patch.

In detail, we leverage the technique in our previous work [23] to generate the semantic summary from a partial code segment (i.e., a partial trace). Here semantics are expressed as the effects of executing the partial code segment on the machine state. The machine state $s$ is characterized by a 3-tuple $\langle Mem, Reg, Flag\rangle$, denoting the memory *Mem*, the general-purpose registers *Reg*, and the condition-code flags *Flag*. The machine state before and after executing the partial code segment is referred to as *pre-state* and *post-state*, respectively. For example, one possible pre-state before executing the code segment in Figure 5(a) is given in Figure 5(b), where all `registers`, `flags`, and `memory` are assigned by the value `0`; and in the corresponding post-state, the registers `eax` and `ebx` hold the values `0x04` and `-0x04`, respectively, while the sign flag `sf` holds the value `1` due to the negative result in `ebx`.

Then, the semantic summary is the difference between the pre-state and post-state, as shown in Equation 2.

$$\mathcal{S} = s_{post} - s_{pre} \qquad (2)$$

For example, the semantic summary of the code segment shown in Figure 5(a) is `eax′ = 0x04` and `ebx′ = eax-0x04`, where primed variables denote final values and non-primed variables denote initial values.

In our semantic analysis, for both patched and original partial traces, we first generate various configurations of pre-state and run the partial traces and measure the corresponding post-state values. Then, we compute the semantic summary for patched and original partial traces, and compare them following the techniques in [24, 23]. Finally, if the semantic difference is below a pre-defined threshold value (i.e., $< \Delta_d$ ). We determine that the patch is a security patch. Otherwise, it is a non-security patch. In our experiment, we empirically fix $\Delta_d$ to be 0.20.

**Example 4:** Let us consider the running example in Figure 1. There are in total 17 machine artifacts involved in the semantic summary computation that are 8 register, 8 status flags and one memory location `[edi+8]`, among which only three status flags (`sf`, `zf` and `cf`) are influenced by the newly added instruction `test eax, eax` in the patched partial trace. Given a fixed pre-state in which all values of artifacts are set to 0 but ebx to 4, as shown in Figure 5(b), the post-state can be computed easily. In the post-state, only the zero flag `zf` is set to 1, while all the other 16 artifacts keep same as the original trace. From the pre-state and post-state, we get

```
mov  eax, 0x04
sub  ebx, eax
```

(a) Sample Code Segment

| Pre-state: | Post-state: |
|---|---|
| $\texttt{Reg} = \{\texttt{eax} = 0, \texttt{ebx} = 4, ..\}$ | $\texttt{Reg}' = \{\texttt{eax}' = \texttt{0x4}, \texttt{ebx}' = \texttt{0x0}, ..\}$ |
| $\texttt{Flag} = \{\texttt{zf} = 0, \texttt{sf} = 0, ..\}$ | $\texttt{Flag}' = \{\texttt{zf}' = 1, \texttt{sf}' = 0, ..\}$ |
| $\texttt{Mem} = \{0, 0 \ldots 0\}$ | $\texttt{Mem}' = \{0, 0 \ldots 0\}$ |

(b) Pre- and Post-State before and after Executing the Code Segment

Fig. 5: An Example of Pre-State and Post-State

that the semantic difference between the patched and original partial trace is equal to $0.058$ ($< \Delta_d$), which implies that the patch in Figure 1 is a security patch.

## D. Summarizing Patch and Vulnerability Patterns

Once the security patches are identified, SPAIN proceeds to summarize patch patterns and the corresponding vulnerability patterns from the original and patched partial traces. To this end, we introduce a light-weight program analysis technique. Specifically, given a patched partial trace $t_p$ and the relevant original partial traces $\{t_o^1, ..., t_o^n\}$, SPAIN identifies the newly-added and security-sensitive instructions in $t_p$. In general, newly-added control transfer instructions, especially the ones that depend on comparison instructions such as `cmp` and `test`, are of the interest for security analysts, which are more likely to reflect the newly-introduced sanity checks in the patch.

In the next step, we pass the source and destination operands of those interesting instructions to the taint engine [25] to track their sources and sinks that are key indicators of vulnerabilities. For example, in the instruction `cmp eax, ebx`, register `ebx` is the source operand and register `eax` is the destination operand. In particular, non-immediate source and destination operands are passed to the taint engine to track their origins (or *sources*) using backward taint analysis, while the non-immediate destination operand is passed to the taint engine to track its destinations (or *sinks*) using forward taint analysis. It is important to note that taint analysis is performed within the patched function, i.e., intra-procedural taint analysis.

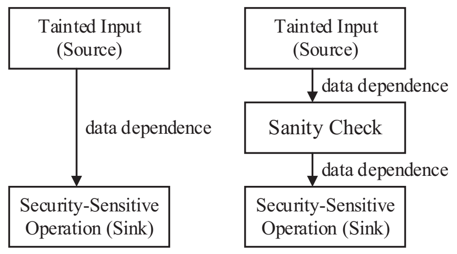
Finally, the tracked sources and sinks are combined to summarize the vulnerability and security patch patterns. In Figure 6, we show the abstract vulnerability and patch patterns, where *source*, *sink* and *sanity check* are defined as follows.

**Definition 4 (Sources):** *Taint sources* are the user/external inputs (i.e., tainted inputs) that can reach the patched function and are used by those interesting instructions.

For example, external function parameters or return values of security-sensitive system APIs (e.g., `scanf`) are considered as taint sources.

**Definition 5 (Sinks):** *Taint sinks* are the security-critical operations that involve the tainted inputs.

For example, memory dereference operations (e.g., `mov eax, [taint-source]`) or arithmetic subtraction operations (e.g., `mov eax, [taint-source]; sub ebx, eax`) that involves taint sources are considered as taint sinks.



(a) Vulnerability Pattern (b) Patch Pattern

Fig. 6: Abstract Vulnerability and Security Patch Pattern

**Definition 6 (Sanity Checks):** *Sanity checks* are operations performed on the tainted inputs before they are involved in security-critical operations.

## E. Applications of Patterns

The patch patterns can be used in many applications. One main application is to search for the similar patches and corresponding vulnerabilities in the binaries. Besides, SPAIN is orthogonal to several patch analysis tools, and hence can provide patch patterns as input for them. For example, Prophet [8] can learn a probabilistic model from the correct code to automatically generate patches. SPAIN can provide possible locations where a patch is needed. TEDEM [26] can identify binary code regions that are similar to code regions containing vulnerabilities. SPAIN can provide such vulnerable regions through pattern matching so that TEDEM may have more candidates to search for. Honey patch [15] was proposed as a trap to monitor attack information and misinform the attacker through redirecting the attack to an unpatched decoy. SPAIN can help to identify the patches that can be converted into honey patches, making the whole process fully automatic. A survey on repeated patches [27] has been conducted to show the general trend of bug fixes. SPAIN can enable the trend analysis on a large number of programs to gain a complete understanding of how programmers fix bugs.

## IV. Evaluation

In this section, we conduct an experimental study on several real-world projects to answer the following research questions.

- **RQ1**: What is the accuracy and scalability of SPAIN to identify security patches?
- **RQ2**: What are the security patch patterns and their corresponding vulnerability patterns summarized by SPAIN?
- **RQ3**: What are the potential application scenarios of the summarized patterns of SPAIN?

The experiments were conducted on an HP z420 workstation with 32GB RAM and Intel Xeon CPU E5-1620 v2 3.70GHz. All the experimental data is available at our website [28]. We used the following real-world software in our evaluation:

- **OpenSSL** is an open source software with around 446,747 number of locations (LOC) and developed since 1998.
- **Linux Kernel** is an open source software with around 18,963,973 LOC and developed since 2002.

TABLE I: Accuracy and Performance on OpenSSL

| Ver. | CVE | Sec. Pat. | | Non-Sec. Pat. | | T.P. | F.P. | T. (s) |
|---|---|---|---|---|---|---|---|---|
| | | G.T. | Iden. | G.T. | Iden. | | | |
| 0–a | 1 | 7 | 4 | 11 | 0 | 0.57 | 0 | 11 |
| a–b | 0 | 0 | 0 | 5 | 0 | – | 0 | 13 |
| b–c | 1 | 3 | 2 | 0 | 0 | 0.67 | – | 7 |
| c–d | 3 | 19 | 8 | 27 | 0 | 0.42 | 0 | 60 |
| d–e | 0 | 0 | 0 | 6 | 1 | – | 0.17 | 8 |
| e–f | 3 | 12 | 9 | 17 | 10 | 0.75 | 0.59 | 14 |
| f–g | 2 | 10 | 8 | 6 | 1 | 0.8 | 0.17 | 11 |
| g–h | 8 | 29 | 18 | 4 | 1 | 0.62 | 0.25 | 20 |
| h– i | 9 | 37 | 24 | 23 | 8 | 0.65 | 0.35 | 54 |
| i–j | 4 | 23 | 11 | 5 | 2 | 0.48 | 0.4 | 35 |
| j–k | 7 | 25 | 19 | 15 | 5 | 0.76 | 0.33 | 39 |
| k–l | 0 | 0 | 0 | 1 | 0 | – | 0 | 7 |
| l–m | 8 | 34 | 25 | 7 | 3 | 0.74 | 0.43 | 95 |
| m–n | 5 | 60 | 49 | 38 | 5 | 0.82 | 0.13 | 76 |
| n–o | 1 | 0 | 0 | 6 | 0 | 0 | 0 | 9 |
| o–p | 2 | 2 | 1 | 1 | 0 | 0.5 | 0 | 9 |
| p–q | 2 | 39 | 31 | 40 | 11 | 0.79 | 0.28 | 48 |
| q–r | 1 | 10 | 9 | 3 | 0 | 0.9 | 0 | 18 |
| r–s | 6 | 13 | 11 | 2 | 0 | 0.85 | 0 | 36 |
| Sum. | 63 | 323 | 229 | 217 | 47 | 0.71 | 0.22 | – |

- **Adobe PDF Reader** is a closed source software. We use two of its libraries, `3difr.x3d` and `AXSLE.dll`, which have around 1,293 and 4,874 functions respectively.

## A. Accuracy and Scalability (RQ1)

*1) Accuracy:* To evaluate the accuracy of SPAIN on identifying security patches, we manually identified all the security and non-security patches of all the 20 versions of OpenSSL 1.0.1 by analyzing its commits on GitHub. They were used as the ground truth to evaluate the true positive and false positive of SPAIN. For each security patch, we also manually analyzed the type of the patched vulnerability.

Table I reports the detailed results of our accuracy evaluation on OpenSSL. The first column lists the version numbers of two consecutive versions of OpenSSL, which are respectively served as the original and patched binaries. The second column reports the number of CVEs that are documented. The third and fifth columns respectively show the number of security and non-security patches we manually identified. The fourth column reports the true security patches SPAIN successfully identified, and the sixth column gives the false security patches SPAIN *incorrectly* identified. Note that SPAIN only reports security patches, here the sixth column gives the false positive cases generated by SPAIN. The last three columns compute the true positive, false positive and execution time of SPAIN.

From Table I it can be seen that among the 323 security patches, SPAIN successfully identified 229 of them, while it incorrectly identified 47 of the 217 non-security patches as security patches. It achieved the true positive rate of 71% and the false positive rate of 22%, which indicated that SPAIN can identify security patches with high accuracy. Besides, compared with the number of identified security patches, only a small number of CVEs are documented, which demonstrates that SPAIN can discover undocumented patches. Moreover, SPAIN took around 32 seconds on average to analyze the binaries. In addition, Table II shows the accuracy of SPAIN with respect to

TABLE II: Accuracy on OpenSSL w.r.t Vulnerability Types

| Vulnerability Type | G.T. | SPAIN | Accuracy |
|---|---|---|---|
| Buffer Overflow | 54 | 37 | 0.69 |
| NULL Pointer Dereference | 65 | 61 | 0.94 |
| Memory Leak | 51 | 30 | 0.59 |
| Double-Free | 14 | 7 | 0.5 |
| Integer Overflow | 8 | 7 | 0.88 |
| Initialization | 16 | 11 | 0.69 |
| Off-by-One | 3 | 2 | 0.67 |
| Side Channels | 2 | 1 | 0.5 |
| Use-After-Free | 1 | 1 | 1 |
| Others | 109 | 72 | 0.66 |
| Sum. | 323 | 229 | 0.71 |

nine vulnerability types. Note that 109 of the security patches patched some tricky vulnerabilities that do not belong to these nine common types. The results indicate that SPAIN can identify security patches for different types of vulnerabilities with high accuracy.

By closely looking into the security patches SPAIN failed to identify, we find two main causes for the false negatives. First, a patch is so simple that our function filtering step may fail to detect the changes. For example, one unidentified security patch in OpenSSL simply increased the buffer size in a function by a constant value. As a result, the patched function is the same to the original one, except for that particular constant. After the basic block normalization, they become the same and will not be further analyzed. Second, a patch is so complicated that our semantic analysis may identify it as a non-security patch due to the large amount of newly-introduced semantics. For example, developers may rewrite part of a function to fix a vulnerability; or some different patches happen to patch the same function. In such cases, our semantic analysis may detect significant semantic difference between the patched function and the original one, failing to identify the security patch.

Similarly, we investigated the causes of the false positives. One main reason is that some non-security patches only slightly modify the program, especially for fixing some performance bugs [29] or adding the consideration for some missed corner cases. For example, a patch added a simple conditional statement that would be executed only when certain criteria have been met. It follows the similar pattern of security patches since most security patches can be seen as a conditional functionality, which redirects the execution to safe places if some variables have unexpected values. Therefore, such kinds of non-security patches are difficult to be distinguished. We argue that false positives only have a small impact on our analysis as the number of false positive is small, and a simple manual validation can identify them.

**Summary.** Based on these observations, we can positively answer RQ1 that SPAIN can identify security patches for different types of vulnerabilities with a high true positive rate as well as acceptable false positive rate. Further, SPAIN can discover security patches that are not documented.

*2) Scalability:* To evaluate the scalability of SPAIN to analyze large binaries, we ran SPAIN on both open source Linux Kernel and closed source Adobe PDF Reader. In particular, we used versions 3.16.2 and 3.16.3 of Linux Kernel, and compiled

TABLE III: Performance on Linux and Adobe PDF Reader

| Version | Total Func. | Sec. Patched | T. (s) |
|---|---|---|---|
| Linux: 3.16.2 – 3.16.3 | 249341 | 1221 | 807 |
| difr.x3d: 11.0.08 – 11.0.09 | 1293 | 12 | 21 |
| difr.x3d: 11.0.13 – 11.0.14 | 1293 | 13 | 23 |
| difr.x3d: 11.0.15 – 11.0.16 | 1293 | 11 | 20 |
| AXSLE.dll: 11.0.15 – 11.0.17 | 4875 | 27 | 84 |

them using the `-o2` optimization level (i.e., the most common commercial setting) with all the functions included. For Adobe PDF Reader, we analyzed the library `3difr.x3d` of versions 11.0.08, 11.0.09, 11.0.13, 11.0.14, 11.0.15 and 11.0.16 as well as the library `AXSLE.dll` of versions 11.0.15 and 11.0.17.

Table III presents the performance of SPAIN on Linux kernel and Adobe PDF Reader. The first column gives the versions of the original and patched binaries, and the second column reports the average number of functions in them. The third and fourth columns list the identified security patches by SPAIN and the corresponding time overhead. We can see that SPAIN analyzed the whole Linux Kernel in 807 seconds, and analyzed the two Adobe libraries in 23 seconds. Note that, because we do not have the ground truth for security patches, we did not show the accuracy of SPAIN on them but reported the identified security patches.

**Summary.** Based on the results from Table III, we can positively answer RQ1 that SPAIN scales well to large binaries.

## B. Patch and Vulnerability Patterns (RQ2)

Table IV presents the vulnerability and patch patterns summarized for the key vulnerability types patched in the 20 versions of OpenSSL. Among them, one of most common ones is double-free vulnerability, i.e., an error that occurs when `free()` is called more than once with the same memory address as an argument. It is summarized in the first row, where the memory address is obtained from an untrusted function as a return value, and it is freed more than once, which leads to the vulnerability. To patch this, one needs to sanity check for validity of the memory address and remove all the occurrences where it is freed for more than once.

The integer overflow/underflow is summarized in the second row, where it occurs when an arithmetic calculation produces a result that is greater (or smaller) in magnitude than that a given register or storage location can store or represent. In general, arithmetic operations are vulnerable to integer overflow or underflow when they take the inputs from untrusted sources and perform some security-sensitive operation such as memory dereferencing and memory indexing on the calculated results. To patch such vulnerable cases, one needs to perform a sanity check on the untrusted inputs before allowing for any security-sensitive arithmetic operation.

The third row summarizes the use-after-free vulnerability pattern, where the obtained pointer (from an untrusted source) to the memory object is freed without checking for liveness property of the pointer. For patching such vulnerabilities, one needs to check whether the object is in use, if so, the pointer to it should not be freed. The fourth row summarizes `NULL` pointer dereference vulnerability, the most frequently occurred vulnerability in OpenSSL. As discussed in Section II, it is very common to obtain a pointer from an untrusted source; hence, before involving it in any security-sensitive operation, such as memory dereferencing, it needs to be checked whether the pointer is `NULL` or valid. Finally, the fifth row summarizes the other most commonly observed vulnerability, buffer overflow or underflow, where the patch suggests that any pointer to a memory object should be properly bounds-checked before involving it in any security-sensitive operation.

Apart from the vulnerability types summarized in Table IV, we observed other class of vulnerabilities/patches that cannot be generalized for pattern matching. Such vulnerability types include, side-channel information leakage, memory leakage and uninitialized variables whose patterns are particular to the OpenSSL binaries. However, summarizing these pattern will enable us to identify clone or copy-paste type vulnerabilities that are very commonly observed in the wild [30, 24]. Due to the space limitation, we provide the vulnerability and patch patterns for such class of vulnerabilities and the patterns for Linux and Adobe in our website [28].

## C. Application of Patterns (RQ3)

Using SPAIN and the summarized patterns, we used pattern matching techniques [23] to discover three zero-day vulnerabilities (CVE-2016-0933, CVE-2016-1037 and CVE-2016-4198) in the two libraries in Adobe PDF Reader.

We start the experiment with an Adobe Reader vulnerability, CVE-2014-0565, which has already been patched in 2014. It is a vulnerability that reads the memory out of bound, due to the insufficient check on the string length of *Line Set* block. Our tool successfully identifies the patch by diffing the two versions of Adobe Reader, 11.0.8 and 11.0.9, i.e., finding the place where two additional checks have been added to the function. After the checks, the program returns to the normal execution. Therefore, our tool identifies security patches with high confidence. It is a very typical patch pattern, which is the buffer overflow pattern in the fifth row of Table IV.

Then, we use the corresponding vulnerability pattern to search for similar vulnerabilities in different versions of Adobe Reader. Specifically, we find CVE-2016-0933 and CVE-2016-1037, having the same vulnerability pattern, which, later, have been confirmed by manual analysis. CVE-2016-0933 is a vulnerability in Adobe Reader 3difr.x3d before version 11.0.14. CVE-2016-1037 also resides in 3difr.x3d, while it is in version 11.0.16. We also find CVE-2016-4198, which shares the similar pattern with the previous vulnerability, although it is indeed an integer overflow vulnerability in Adobe XSLT library. Once triggered, it will cause an out of bound write to the memory and remote code execution.

After Adobe Reader had patched the aforementioned vulnerabilities, we used SPAIN to diff the patched version with the original vulnerable version. We successfully located the three vulnerability patches using the tool, as shown in Table III, which demonstrates that our tool has the capability to capture patches in closed source binaries. A more detailed explanation of the patterns can be found at our website [28].

TABLE IV: Patch Patterns vs. Vulnerability Patterns, where TNT denotes taint, IN denotes input, and SSTV denotes sensitive.

| Vul. type | Concrete Vulnerability | Vulnerability Pattern | Concrete Patch | Patch Pattern |
|---|---|---|---|---|
| Double Free | call BN_to_ASN1_INTEGER<br>test eax, eax<br>mov esi, eax<br>...<br>mov [esp + 4Ch + var_4C], esi<br>call ASN1_STRING_clear_free<br>...<br>mov [esp + 4Ch + var_4C], esi<br>call ASN1_STRING_clear_free | call ⟨untrusted_func⟩<br>⟨sanitycheck⟩ : ⟨return_value⟩<br>...<br>mov ⟨func_param⟩, ⟨return_value⟩<br>call ⟨free⟩<br>...<br>mov ⟨func_param⟩, ⟨return_value⟩<br>call ⟨free⟩ | call BN_to_ASN1_INTEGER<br>test eax, eax<br>mov esi, eax<br>...<br>mov [esp + 4Ch + var_4C], esi<br>call ASN1_STRING_clear_free<br>... | call ⟨untrusted_func⟩<br>⟨sanity check⟩ : ⟨return_value⟩<br>...<br>mov ⟨func_param⟩, ⟨return_value⟩<br>call ⟨free⟩<br>... |
| Integer Underflow Overflow | mov eax, [esp + 4Ch + arg_4]<br>...<br>mov ecx, eax<br>sub ecx, [esp + 4Ch + var_2C]<br>add [esp + 4Ch + var_24], ecx<br>...<br>mov esi, [esp + 4Ch + _24]<br>mov ecx, [esp + 4Ch + arg_8]<br>mov [ecx], esi | mov ⟨TNT_IN⟩, ⟨untrusted_src⟩<br>...<br>⟨arith_op⟩ ⟨TNT_result⟩, ⟨TNT_IN⟩<br>...<br>mov ⟨sec_SSTV_sink⟩, ⟨TNT_result⟩ | mov eax, [esp + 4Ch + arg_4]<br>...<br>mov edi, [esp + 4Ch + var_2C]<br>cmp eax, edi<br>jl error<br>mov ecx, eax<br>sub ecx, [esp + 4Ch + var_2C]<br>add [esp + 4Ch + var_24], ecx<br>...<br>mov esi, [esp + 4Ch + _24]<br>mov ecx, [esp + 4Ch + arg_8]<br>mov [ecx], esi | mov ⟨TNT_IN⟩, ⟨untrusted_src⟩<br>⟨sanity check⟩ : ⟨TNT_IN⟩<br>...<br>⟨arith_op⟩ ⟨TNT_result⟩, ⟨TNT_IN⟩<br>...<br>mov ⟨sec_SSTV_sink⟩, ⟨TNT_result⟩ |
| Use After Free | mov ebp, [esp + 0ECh + arg_0]<br>...<br>mov [esp + 0ECh + dest], ebp<br>call ssl3_release_read | mov ⟨TNT_IN⟩, ⟨untrusted_src⟩<br>...<br>mov ⟨func_param⟩, ⟨TNT_pointer⟩<br>call ⟨free⟩ | mov ebp, [esp + 0ECh + arg_0]<br>...<br>mov eax, [ebp + 58h]<br>mov eax, [eax + 0F8h]<br>test eax, eax<br>jnz ⟨do not release⟩<br>...<br>mov [esp + 0ECh + dest], ebp<br>call ssl3_release_read | mov ⟨TNT_IN⟩, ⟨untrusted_src⟩<br>⟨sanity check⟩ : ⟨TNT_pointer⟩<br>...<br>mov ⟨func_param⟩, ⟨TNT_pointer⟩<br>call ⟨free⟩ |
| NULL Pointer Dereference | callBN_new<br>mov edi, eax<br>...<br>cmp eax, [edi + 8] | call⟨untrusted_func⟩<br>...<br>⟨mem_deref⟩ : ⟨return_value⟩ | call BN_new<br>test eax, eax<br>mov edi, eax<br>jz error<br>...<br>cmp eax, [edi + 8] | call⟨untrusted_func⟩<br>⟨sanity check⟩ : ⟨return_value⟩<br>...<br>⟨mem_deref⟩ : ⟨return_value⟩ |
| Buffer Overflow | mov ebx, [esp + 3Ch + arg_8]<br>...<br>mov [esp + 3Ch + n], ebx<br>call eax | mov ⟨TNT_IN⟩, ⟨untrusted_src⟩<br>mov ⟨func_param⟩, ⟨TNT_IN⟩<br>...<br>call ⟨untrusted_func⟩ | mov ebx, [esp + 3Ch + arg_8]<br>cmp ebx, [esp + 3Ch + arg_4]<br>jl error<br>...<br>mov [esp + 3Ch + n], ebx<br>call eax | mov ⟨TNT_IN⟩, ⟨untrusted_src⟩<br>⟨sanity check⟩ : ⟨TNT_IN⟩<br>...<br>mov ⟨func_param⟩, ⟨TNT_IN⟩<br>call ⟨untrusted_func⟩ |

## V. Discussion

Our framework has the following limitations. First, our framework tries to look for patches and vulnerabilities with patterns. However, some real-world vulnerabilities are actually the corner cases, which have unique signatures and patches may fix bugs in unusual ways. SPAIN may not be able to achieve high accuracy when dealing with these special cases, in which almost all the patches are usually complex.

Second, vulnerabilities may be patched in some places other than the places where they are triggered. Therefore, trying to hunt them from patches may not be straightforward. Our current tool can only look for vulnerabilities that are patched within the one function where the patch has been found. In future, we plan to adopt some other techniques such as function summarization, more advanced slicing and symbolic execution to handle the vulnerabilities that are patched across functions.

Third, building a general solution for all kinds of binary architectures requires lifting the instruction into intermediate representation. We did not do that because we want to obtain the exact patterns of the patches and vulnerabilities. Binary lifting may result in loss of a certain amount of information or inaccuracy [31], which may be critical for our pattern summarization. Therefore, throughout this work, all the binaries used are in X86 32bit format. It is worth mentioning that our approach is general and can be adapted to other binary formats.

Fourth, the framework shares the common drawback of static vulnerability searching approaches. Although we can identify the location of the vulnerability, we cannot reproduce it in real world if the program is too complicated. The user input may travel through many functions and be transformed several times until it reaches the location that triggers the vulnerability. Hence, we may not have a concrete proof-of-concept (POC) to validate the vulnerability whether it has a real impact on the program's security. However, unlike other searching methods, since we are searching the vulnerability based on its patch, we have a relatively high confidence to say that it is a severe bug, which is needed to be patched.

## VI. Related Work

Our work attempts to understand patches and the corresponding vulnerabilities, and summarize their patterns for discovering similar patches or vulnerabilities. Hence, we discuss the related work in the areas of patch analysis and vulnerability modeling.

### A. Patch Analysis and Diffing

PVDF [32] computes the semantic of patches for privilege elevation vulnerabilities. The patch semantic is then used to guide fuzzy testing to discover new vulnerabilities in binary programs. It takes the vulnerable binary and the corresponding patch as the inputs, and leverages forward and backward taint analysis to extract the semantics. This work is similar to ours; but it assumes the availability of patches, and only focuses on one particular vulnerability type. Differently, SPAIN attempts to summarize patterns for different vulnerability types, and only requires the binary programs but not the patches.

BISSAM [14] uses binary patch diffing information to develop signatures for the security update. Then it executes the binary with malicious inputs under dynamic monitoring to obtain the execution paths. It identifies the vulnerability by determining whether the path includes one of the signatures. Its effectiveness heavily relies on the malicious inputs and assumes the availability of security patches. Differently, we can automatically identify the security patches in a static way.

BinHunt [13] can automatically find the semantic differences in binary programs. It lifts the binary instructions to Intermediate Representation (IR), and constructs CFGs at the IR level. Then it uses graph isomorphism to find the differences and performs symbolic execution on them to obtain the semantics for theorem proving. Unlike it, SPAIN is much more scalable, since we apply function-level filtering, which enables us to search for patches in real-world programs.

Apart from these binary-level patch analysis, there are many source-level approaches. Tian et al. [11] attempt to identify Linux patches based on the commit messages and the source code diffing analysis. They extract features and leverage machine learning techniques to predict whether a commit is a bug fix or not. Soto et al. [33] investigate Java projects to get a deeper understanding of each patch's signature to guide automatic program repairing. BugTrace [12] aims at building the link between the bug and the fix through patch analysis. Kim and Notkin [34] build a diffing tool to infer the structural differences between codes, which helps programmers to discover bugs by comparing two versions of a program. These works all use patch analysis to gain the understanding at the source code level, which may not work if the source code is not available. Instead, SPAIN directly works at the binary level.

## B. Vulnerability Modeling and Searching

INDIO [35] leverages symbolic execution to analyze binaries and detects integer overflow vulnerabilities. It uses some heuristic patterns to find the potential vulnerability candidates. Then it ranks the vulnerable possibility for the candidates. Finally, it selectively executes symbolic execution to remove the false positives further. It discovered known CVEs as well as unknown integer overflow vulnerabilities in real world Window binaries. Similarly, IntScope [36] employs symbolic execution to detect integer overflow vulnerabilities in X86 binaries, and Firmalice [37] uses symbolic execution to detect authentication bypass vulnerabilities in firmware.

Brumley et al. [38] showed that automatic patch-based exploit generation (APEG) was possible. They combine dynamic symbolic execution and static control flow graph analysis to summarize the constraint formulae of the vulnerabilities, which were successfully used to generate exploits for 5 real-world vulnerabilities.

GUEB [39] searches for use-after-free vulnerability patterns in the binary programs. It builds an abstract memory model for the binary functions. Then it uses value set analysis to reason each variable in the assignment and free instructions. If a variable is used after the free instruction, GUEB reports it as a vulnerability. It found one real-world use-after-free vulnerability in ProFTPD program. LoongChecker [40] also uses value set analysis to statically detect potential vulnerabilities in binaries.

The aforementioned works leverage program analysis methods to extract semantics of the binary program, and compare them with certain models to discover potential vulnerabilities. They require expertise to build such models specifically for a specific type of vulnerabilities. Differently, our work aims at automatically summarizing the patterns for different types of vulnerabilities and using them to search for vulnerabilities.

Apart from these binary-level vulnerability modeling and searching approaches, there are many works targeting vulnerability modeling at the source code level. Yamaguchi et al. [41] use a taint-style pattern to search vulnerabilities and filter out irrelevant code. It can reduce the code base by 94.9% on average to improve the code audit efficiency. Wagner et al. [42] reduce the searching of the buffer overflow vulnerabilities to an integer range analysis problem, which discovered vulnerabilities in the Sendmail software. Averinos et al. [43] proposed an automatic exploit generation tool and analyzed 20 open source programs. They use preconditioned symbolic execution to generate control flow hijack attacks and discovered 2 unknown vulnerabilities. However, those approaches work on the source code level; but they fail to analyze binary-level patches. Moreover, more attention has been paid to address the gap between the source code and the compiled binaries [44]. The compiler will introduce bugs even if the source code is correct. Therefore, we choose to work on binary level to be closer to the machine so that, ideally, the framework can capture all possible vulnerabilities.

## VII. Conclusion

In this paper, we proposed a patch analysis framework SPAIN to automatically learn the security patch patterns and vulnerability patterns, and identify them from the program binary executables. It has built the bridge from the binary diffing to the automatic patch understanding. The experiments have shown that SPAIN can correctly locate more than half of the vulnerability patches in the binaries and find n-day vulnerabilities in major commercial software. SPAIN can be useful in vulnerability and patch understanding, similar bug hunting, binary code auditing, and, eventually, the program security enhancement. In the future, we plan to extend the framework to generate more detail and precise report of each patch and vulnerability through program slicing and symbolic execution, as well as to make it capable of analyzing binaries with other instruction sets, like ARM. Also, we would like to set up binary vulnerability databases with the help of the tool.

## VIII. Acknowledgements

This research has been supported in part by the National Research Foundation, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30), and partially been sponsored by the National Science Foundation of China (No. 61572349, 61272106, 61402179).

## References

[1] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *ICSE*, 2009, pp. 474–484.

[2] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *S&P*, 2015, pp. 725–741.

[3] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.

[4] C. Hu, Z. Li, J. Ma, T. Guo, and Z. Shi, "File parsing vulnerability detection with symbolic execution," in *TASE*, 2012, pp. 135–142.

[5] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "Its4: a static vulnerability scanner for c and c++ code," in *ACSAC*, 2000, pp. 257–267.

[6] J. Heffley and P. Meunier, "Can source code auditing software identify common vulnerabilities and be used to evaluate software security?" in *HICSS*, 2004, pp. 90 277–.

[7] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *S&P*, 2017.

[8] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *POPL*, 2016, pp. 298–312.

[9] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*, 2013, pp. 802–811.

[10] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *ICSE*, 2015, pp. 448–458.

[11] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *ICSE*, 2012, pp. 386–396.

[12] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, "Recovering traceability links between source code and fixed bugs via patch analysis," in *TEFSE*, 2011, pp. 31–37.

[13] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *ICICS*, 2008, pp. 238–255.

[14] T. Schreck, S. Berger, and J. Göbel, "Bissam: Automatic vulnerability identification of office documents," in *DIMVA*, 2013, pp. 204–213.

[15] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation," in *CCS*, 2014, pp. 942–953.

[16] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *USENIX Security*, 2015, pp. 611–626.

[17] Hex-Rayd, "IDA Pro," http://www.datarescue.com/idabase, Last 2016: Aug. 2016.

[18] H. Flake, "Structural comparison of executable objects," in *DIMVA*, 2004, pp. 161–173.

[19] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *USENIX Security*, 2016, pp. 583–600.

[20] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *ICSE*, 2014, pp. 175–186.

[21] 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 *FSE*, 2014, pp. 389–400.

[22] Intel, "Intel 64 and ia-32 architectures software developer's manual," http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, Last 2016: Aug. 2016.

[23] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: cross-architecture cross-os binary search," in *FSE*, 2016, pp. 678–689.

[24] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *S&P*, 2015, pp. 709–724.

[25] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," in *ACSAC*, 2006, pp. 269–278.

[26] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *ACSAC*, 2014, pp. 406–415.

[27] J. Park, M. Kim, B. Ray, and D. H. Bae, "An empirical study of supplementary bug fixes," in *MSR*, 2012, pp. 40–49.

[28] SPAIN, http://pat.scse.ntu.edu.sg/spain/.

[29] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *ICSE*, 2015, pp. 902–912.

[30] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 359–368.

[31] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, *BAP: A Binary Analysis Platform*, 2011, pp. 463–469.

[32] S. Letian, F. Jianming, C. Jing, and P. Guojun, "PVDF: An automatic Patch-based Vulnerability Description and Fuzzing method," in *CSC*, 2014, pp. 1–8.

[33] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo, "A deeper look into bug fixes: Patterns, replacements, deletions, and additions," in *MSR*, 2016, pp. 512–515.

[34] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE*, 2009, pp. 309–319.

[35] Y. Zhang, X. Sun, Y. Deng, L. Cheng, S. Zeng, Y. Fu, and D. Feng, "Improving accuracy of static integer overflow detection in binary," in *RAID*, 2015, pp. 247–269.

[36] T. Wang, T. Wei, Z. Lin, and W. Zou, "Intscope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution," in *NDSS*, 2009.

[37] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *NDSS*, 2015.

[38] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *S&P*, 2008, pp. 143–157.

[39] J. Feist, L. Mounier, and M. Potet, "Statically detecting use after free on binary code," *J. Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.

[40] S. Cheng, J. Yang, J. Wang, and F. Jiang, "Loongchecker: Practical summary-based semi-simulation to detect vulnerability in binary code," in *TrustCom*, 2011, pp. 150–159.

[41] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *S&P*, 2015, pp. 797–812.

[42] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *NDSS*, 2000.

[43] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: automatic exploit generation," in *NDSS*, 2011.

[44] V. D'Silva, M. Payer, and D. X. Song, "The correctness-security gap in compiler optimization," in *SPW*, 2015, pp. 73–87.