**PATCHScope: Memory Object Centric Patch Diffing**

Lei Zhao†
leizhao@whu.edu.cn
Wuhan University

Yuncong Zhu
yuncongzhhu@whu.edu.cn
Wuhan University

Jiang Ming†
jiang.ming@uta.edu
University of Texas at Arlington

Yichen Zhang
EddenZhang@outlook.com
Wuhan University

Haotian Zhang
haotian.zhang@mavs.uta.edu
University of Texas at Arlington

Heng Yin
heng@cs.ucr.edu
University of California, Riverside

**ABSTRACT**

Software patching is one of the most significant mechanisms to combat vulnerabilities. To demystify underlying patch details, the techniques of patch differential analysis (a.k.a. patch diffing) are proposed to find differences between patched and unpatched programs’ binary code. Considering the sophisticated security patches, patch diffing is expected to not only correctly locate patch changes but also provide sufficient explanation for understanding patch details and the fixed vulnerabilities. Unfortunately, none of the existing patch diffing techniques can meet these requirements.

In this study, we first perform a large-scale study on code changes of security patches for better understanding their patterns. We then point out several challenges and design principles for patch diffing. To address the above challenges, we design a dynamic patch diffing technique PATCHScope. Our technique is motivated by two key observations: 1) the way that a program processes its input reveals a wealth of semantic information, and 2) most memory corruption patches regulate the handling of malformed inputs via updating the manipulations of input-related data structures. The core of PATCHScope is a new semantics-aware program representation, memory object access sequence, which characterizes how a program references data structures to manipulate inputs. The representation can not only deliver succinct patch differences but also offer rich patch context information such as input-patch correlations. Such information can interpret patch differences and further help security analysts understand patch details, locate vulnerability root causes, and even detect buggy patches.

**CCS CONCEPTS**

- Security and privacy → Software reverse engineering.

1. **INTRODUCTION**

Software patching is one of the most significant mechanisms to combat vulnerabilities [1]. Security patches imply abundant information about the fixed vulnerabilities, which benefits the construction of hot patches [54, 59], input filters [14, 39], and honey patches [2, 4]. Besides, by extracting patch signatures or semantics, researchers propose to test the presence of patches [74], detect vulnerabilities with patch-enhanced signatures [69], and evaluate the security impacts of patches [68]. On the offense side, patch details promote the study of attack vectors and potential variants of patched vulnerabilities [8, 31, 50], such as “1-day” exploit generation [8, 47, 50, 55, 62] and “buggy patch” [3, 24] detection.

Despite the significance of patches, demystifying patch details, especially for commodity software, is challenging. Commodity software vendors often release patches in binary format. Without the support of patch source, it is difficult to understand patch details or even locate patch-relevant code changes in binaries. Although various online vulnerability databases (e.g., CVE, NVD) archive identified vulnerabilities, most patch or vulnerability bulletins are ambiguously worded on technical details [13, 48]. These bulletins often report simple information such as vulnerability type and security impacts [46], which is far away from investigating vulnerability root causes and understanding patch details.

**Patch analysis at the binary level.** Given a patched program and an unpatched one, patch analysis aims to locate differences induced by patches and interpret such differences with rich semantics, for investigating the root causes of the fixed vulnerability, as well as understanding patch details such as how the patch fixes the vulnerability. To ease presentation, we denote comparison-based patch analysis as patch diffing.

To locate patch-relevant code changes, an intuitive approach is to apply binary diffing techniques [27, 34, 49] to capture code differences, as differences may indicate the changes caused by patches. The basic insight behind binary diffing techniques is to define code similarity representations, ranging from static features (e.g., control
A more challenging task in patch analysis is to interpret identified patterns. We classify security patches into nine categories in terms of code changes induced by security patches to understand their impact.

Our approach. In this study, we first perform a large-scale study on code changes induced by security patches to understand their patterns. We classify security patches into nine categories in terms of their code changes. By inspecting code changes, we observe that patterns used in existing studies are incomplete to characterize complicated types of patches. Further, we point out several challenges and design principles for patch analysis at the binary level.

To address challenges in patch analysis, we design a dynamic patch diffing technique, named PATCHSCOPE. Compared with existing techniques that represent identified differences as low-level instructions, PATCHSCOPE represents patch differences as a higher-level abstraction with rich semantics, which can facilitate the investigation on the root causes and patch details.

The core of PATCHSCOPE is a semantics-aware program representation: memory object access sequence. Our approach is motivated by two key observations: 1) the way that a program processes its input reveals a wealth of semantic information; 2) most security patches, especially for memory corruption vulnerabilities, aim to better regulate the handling of bad inputs. More specifically, when receiving an input, a program subsequently parses the received input into multiple fields and references various data structures to manipulate the input.

In detail, we dynamically excavate program memory objects that are referenced during execution based on memory access patterns. With multi-tag taint analysis, we further identify input fields that are correlated with corresponding memory objects. Then, we define a memory object access model to represent the manipulations on input-related data structures. Therefore, given a PoC, we identify patch differences via the comparison between the memory object access sequences from the two executions on the unpatched and patched programs respectively.

We evaluate the performance of PATCHSCOPE and existing binary diffing techniques on real-world vulnerabilities. Evaluation results show PATCHSCOPE delivers more concise and accurate results than existing binary diffing techniques. Further, we detect several cases that their security patch differences only reveal at the level of memory object access sequence. To demonstrate that PATCHSCOPE can facilitate understanding patch details, we summarize how differences in memory object access sequence reflect the impacts of patches, and present cases for dealing with complicated patches such as overwriting an entire function. At last, we find a “buggy patch” case that the patched vulnerabilities are not completely repaired, leaving end-users still vulnerable.

Contributions. In summary, we make the following contributions:

- **New insights from a large-scale study on security patches.** We perform a large-scale investigation on security patches from the perspective of code changes, which is less studied in previous literature. By inspecting the source code of patches, we characterize security patches into nine categories with respect to their code changes. Further, we study the impact of patch patterns on code changes, and point out that several challenges in locating patched code at the binary level. Our investigation can promote studies on patch analysis.
• A memory-object-centric technique for patch diffing. We propose a new angle to identify patch differences in terms of how programs manipulate inputs via corresponding data structures. Our semantics-aware approach is not only robust for complicated types of patches, but also can deliver rich semantics that other tools cannot offer. Such information can assist security experts in patch analysis and free the burden of manually reverse engineering efforts.

• Effectiveness in patch analysis. Evaluation on real-world vulnerabilities shows that PatchScope outperforms existing patch diffing techniques with more concise and accurate results of located patch differences. Case studies demonstrate that PatchScope can further interpret patch differences with rich semantics delivered by memory object access sequence.

2 BACKGROUND AND MOTIVATION

In this section, we describe the problem setting, characterize security patch patterns via a large-scale study, and demonstrate the limitations in existing binary diffing techniques.

2.1 Problem Setting

We use $P$ and $P'$ to denote a vulnerable program and the patched version containing a security fix, respectively. Both $P$ and $P'$ are stripped binaries. Their debug information and symbols are missing. By comparing the execution traces of $P$ and $P'$ on the same PoC, our goal is to 1) locate patched differences and 2) interpret identified differences with rich semantics, for understanding patches as well as fixed vulnerabilities.

Our study explores the direction of dynamic patch diffing. Thus, we assume a PoC is available. With our problem setting, how to obtain or generate a PoC is application dependent and out of scope. Actually, obtaining or generating a PoC is challenging and may cost significant manual effort [10, 46, 73]. As we will show, dynamic patch diffing with a PoC is a common yet challenging practice.

For one thing, crash analysis on a PoC is a common practice for security analysts. When a security patch is released, only brief information, such as vulnerability types and security impacts [46] is indicated in vulnerability bulletins, which is far away from technical details. In such a scenario, security analysts often try to seek or generate a PoC for further dynamic analysis. Take the vulnerability CVE-2014-6332 for illustration. Microsoft released the patch on Nov. 11th 2014, with little information about technical details. Later, a security researcher releases a PoC [61]. After that, multiple technical reports present the vulnerability details via dynamic analysis on the released PoC.

Even with a PoC, dynamic execution analysis is still challenging, especially on binaries where high-level program abstractions are missing. This process has been proven to be difficult and tedious [30, 67], which may take a significant amount of time and domain knowledge, due to types of vulnerabilities and their complexities [6].

2.2 Security Patch Patterns

To understand security patch patterns, we perform a large-scale study by manually analyzing security patches from five recent studies [37, 41, 46, 65, 68]. These five datasets cover patches for a wide range of vulnerabilities in open-source programs. For our purpose, we only select patches for memory corruption vulnerabilities and exclude duplicate patches in these five datasets. In total, we investigate 2,205 security patches.

According to the definition of patch patterns in previous studies [45, 66], we summarize these security patches into nine categories in Table 1 in terms of their code changes. Please note that a security patch may be counted into more than one category if it introduces multiple change types. From the security patch patterns in Table 1, we have two observations.

Types of patch patterns. First, security patch patterns are of sophisticated types. As shown in Table 1, the top two categories involve adding or updating input sanitization checks, as they are the most direct ways to block unsafe inputs. The types of No. 3, 4, and 5 augment the data structures that are used to reference input data, including changing the data type of int to unsigned int, resizing a buffer, and allocating a heap memory segment rather than a buffer in the stack to receive program inputs. The types of No. 6, 7, 8, and 9 are related to fixing vulnerable functions and their parameters (e.g., replacing unsafe C library functions). These results confirm that a non-trivial proportion of software vulnerabilities are caused by specific program logic faults, and how to fix a vulnerability will consequently depend on program logic.

Previous patch analysis [72] and “1-day” exploit generation techniques [8, 50] propose to identify security patches based on patterns. These techniques mainly focus on the top two categories in Table 1, but fail to analyze other types of patches. Figure 1 shows an example. This patch tries to fix a buffer overflow vulnerability by changing data structures, and it involves only 4 LOC changes within a single function. In particular, two local arrays with fixed lengths are updated to two dynamically allocated heap memory objects. Unlike classical patches that will block bad inputs, this patch fixes the vulnerability by updating program data structures.

Impact on Code Changes. Second, security patch patterns indicate two extremes regarding the binary code difference effects. At one end, 71% of them (e.g., No. 1, 7, 8, and 9 in Table 1) explicitly change the control flow graph (CFG) or call graph (CG): they add new branches, basic blocks, or functions. At the opposite end, the rest of the security patches do not break the integrity of CFG/CG structures but only cause intra-basic-block differences.
int main() { …; serveconnection(sockfd); …; }

int serveconnection(int sockfd) {
    char tempstring[8192]; …
}

Figure 1: A security patch aims to fix a buffer overflow vulnerability in ghttpd-1.4.3. Line 1–4 (main and serveconnection) are shared by two versions. The patch updates two vulnerable data structures from stack memory to heap memory.

Some patches (e.g., increasing the size of a buffer) even leave no evidence on the change of CFG, abstract syntax tree, or program dependency graph. In particular, some security patches may cause no basic-block or instruction differences. For example, a security patch fixes a format string from %s to %%9s. As the format string belongs to the .RODATA section of a binary, this fix will induce no instruction differences.

### 2.3 Limitations of Existing Work

In the binary diffing literature, there are four types of problem scopes: literally identical, syntactically equivalent, semantically similar, and slightly modified [18]. Code representations used for measure similarity or identify differences, ranging from syntactic (e.g., control flow/call graph [7, 25, 29]) to semantics-aware features (e.g., dynamic behaviors [20] and system call sequences [5, 64]), are specific to different problem scopes and vary a lot. Considering sophisticated types of patch patterns and their impact on code changes, we will discuss limitations in existing techniques in the following sections.

**Syntax-based binary diffing.** BinDiff [27], Diaphora [34], and DarunGrim [49] are three industry-standard binary diffing tools with wide applications [31, 47, 48, 55, 62]. These techniques compute the similarity with a set of heuristics on CFG/CG structures, basic blocks, and instructions.

However, these binary diffing tools may not be robust or accurate. The small degree of code changes, such as resizing a buffer, updating the variable type, changing function parameters, may exhibit no difference in assembly code. On the opposite side, some patches involve a large degree of code changes such as overwriting a function. Then, they will generate a large number of low-level code differences that plague security experts [40]. Besides, recent studies also show that binary diffing techniques may raise many false positives [40].

Taking advantage of run-time information, dynamic execution comparison [30] can filter irrelevant code changes that are not traversed along with the execution, which is more robust than static binary diffing. The main limitation is execution comparison may result in a large number of different instructions, and such low-level differences in instructions or operand values cannot deliver rich semantics for understanding patch details and fixed vulnerabilities.

For the example in Figure 1, BinDiff reports up to 30 differences in terms of the instructions removed or added. Due to the page limit, we list a snippet of the instruction alignment sequence in Appendix A. We can observe that it is far from meeting the goal of patch analysis. First, BinDiff did an inaccurate trace alignment. Second, low-level differences cannot avail security experts much regarding how to trigger this vulnerability. We will continue to elaborate on this example in the later sections, but for now, we remind the reader that Figure 1 (b) is a “buggy patch”. With the help of PatchScope, we find this patch induces a new attack vector by overrunning the objects in the heap.

**Symbolic execution for binary diffing.** Another line of research employs symbolic execution for binary diffing, either at the source code level [35, 51, 52] or the binary level [15, 25, 40, 70]. These techniques perform symbolic execution to represent the code snippets as formulas and then detect differences using a theorem prover.

There are several challenges when applying symbolic-execution-based binary diffing for patch analysis. For one thing, most of these techniques rely on static analysis to locate code differences. DSE [51] and DiSE [52] leverage static analysis techniques to identify differences, and then perform symbolic execution to characterize different program behaviors or effects. As we demonstrate above, binary diffing tools may not be robust or accurate to locate patch-relevant changes. Second, the output of symbolic execution—symbolic formulas, especially formulas generated from instructions, is difficult to understand the effects of patches. At last, symbolic execution is typically performed within a basic block [15, 25, 40] or a loop body [70] at the binary level, which may not be scalable to deal with complicated patches involving library function calls (e.g., No. 6, 7, 8, and 9 in Table 1).

**Semantics-aware binary diffing.** Semantics-aware binary diffing techniques leverage system call libraries, or library API functions to represent program semantics, which are commonly adopted by clone detection [20, 64] and malware variant comparison [5, 23, 33, 44]. However, such code representations are often too coarse-grained to be sensitive for patch differences, or because many security patches only induce a small degree of code changes. Another type of semantics-aware technique treats a pair of binary code snippets [20, 63] as a BlackBox and perform dynamic testing to compare their behaviors for similarity measurement. For example, BLEX [20] collects dynamic memory accesses for similarity measurement. However, dynamic memory accesses are too fine-grained to precisely identify patch differences, as BLEX [20] reports 32 differences for the example in Figure 1 (as shown in Appendix B).
AI-powered binary diffing. Taking advantages of AI techniques (e.g., deep learning and neural networks), AI-powered binary diffing techniques [18, 19, 26, 38, 42, 43, 71, 76] first translate binary code snippets into a set of feature vectors, and then they apply machine learning algorithms to the similarity calculation. When handling security patches, this new trend encounters similar problems with the above semantics-aware binary diffing: insensitive to a small degree of code changes.

2.4 Design Principles
To overcome limitations in existing patch analysis techniques, we design patch diffing to satisfy three unique principles.

- For types of patch patterns, patch diffing should be robust in locating patch differences beyond assembly code. Otherwise, it either falls short of capturing small patch differences or suffers from too many code differences.
- Patch diffing should output interpretable results by providing detailed patch relevant differences.
- Patch diffing should deliver rich semantics, especially high-level program representations, for understanding patch details and fixed vulnerabilities.

3 MEMORY OBJECT ACCESS SEQUENCE
In this study, we design a new code representation, memory object access sequence (MOAS), to represent program semantics and compare MOAS for patch diffing. MOAS is motivated by two key observations.

3.1 Key Observations
First, the way that a program references data structures to manipulate inputs can be regarded as a "side effect" of the program semantics. To process a received input, a program typically manages multiple data structures (e.g., variable, array, and struct) to manipulate these fields. For example, "temp[200]" and "*temp" in Figure 1 are defined to store a string that contains two fields: an Internet host address and an HTTP request URL. Therefore, input manipulations via various data structures reflect a wealth of program semantics information. At the binary level, compilers allocate various memory objects to represent high-level program data structures. Then, referencing a data structure to manipulate an input field becomes a memory object access (MOA). The formal definition of MOA is given next.

Second, to fix memory corruption vulnerabilities, most of the security patches add new or update operations on input manipulations for handling bad inputs. Then, security patches will introduce different memory object access between the dynamic executions on P and P'. For example, the patches related to input sanitization checks add or update path conditions to block unsafe inputs, and thus they will introduce subtraction or a bit-wise logical and operations to certain memory objects.

3.2 Semantics-aware Program Representation
Definition 1: Memory Object. At the binary level, we use a memory object to represent a reverse-engineered data structure along an execution trace. A memory object is denoted as mobj = (alloc, size, type). alloc refers to the context information when allocating the memory object, size indicates the size (in byte) of the memory object, and type means its type. In our study, type includes static variables, dynamically allocated variables in the heap, and local variables in the stack. As memory object allocations vary from different types of data structures, the definition of alloc has to be considered together with the program context where mobj is allocated.

- For a static variable, as its memory slot will not be reused, its memory address can exclusively represent alloc.
- For a local variable in the stack, a function frame generally holds the set of local variables. Thus, we represent alloc as a pointer addressed by the frame pointer, as well as the calling context of this function. An exception is register allocation optimization, which assigns local variables into registers to reduce the number of stack memory accesses. Our solution is to use registers to represent such special local variables.
- For a dynamic variable in the heap, we hook the invocation of memory-allocation functions, and then we use the calling site and its return value (the pointer to the allocated heap object) to represent alloc.

We illustrate alloc using our running example from Figure 1. The function Log defines a local variable temp with 200-bytes length. For the dynamic execution on ghttpd-1.4.3, the alloc for temp will include the calling context, main-serveconnection-Log, and the offset from the frame pointer of Log. As ghttpd-1.4.4 defines temp as a heap variable, the alloc for temp includes the calling context, main-serveconnection-Log, the malloc call, and the returned pointer to temp. We will discuss how to identify size in §4.4.

Definition 2: Memory Object Access. A memory object access is denoted as $A(mobj) = (mobj, cc, op, optype, α)$, where:

- mobj is a memory object as defined by Definition 1.
- cc records the context in which mobj is used. As mobj can be accessed by other functions through pointers, cc includes the function that references mobj as well as the calling context of this function.
- The element op refers to the data-flow-related operations on mobj. We consider three main kinds of instructions that participate in input propagation: data movement instructions, arithmetic instructions, and calling instructions to library/system calls. We denote op as the opcodes of these instructions as well as their addresses.
- The element optype contains two access types of mobj: read and write.
- The element α refers to the consecutive bytes of an input that is correlated with mobj during an execution. Please note that α may not be persistent with an input field defined by syntactical formats, because a program subsequently parses and manipulates inputs step by step.

3.3 MOAS Comparison
The temporal order of memory object accesses (MOA) along an execution trace forms a memory object access sequence (MOAS). With the above definitions, we monitor the execution of P and P' on the same PoC. Then we dynamically collect two memory object
We take the top left item in (a2) as an example to interpret memory object access (mobj, cc, op, optype, α). “L,” (mobj) represents a memory object defined in (a1); “<main-serveconnection-Log>” (cc) is the function that references “L1”; “0x804a35c:call vsprintf” (op) refers to the instruction operating on “L1”; “R” (optype) means the access type of read; “[0x4,0x01d]” (α) indicates the input field that is correlated with “L1”.

Figure 2: MOAS comparison result for the patch differences shown in Figure 1. We highlight the misalignment places. “0x65” in b1 indicates that the value of size is tainted.

(b1) Memory object representation for ghttpd-1.4.4

(b2) Memory object access sequence for ghttpd-1.4.4

access sequences from P and P’, respectively. At last, we detect their differences for patch analysis.

According to our observations in Section 3.1, a security patch can leave a fingerprint on one of the following features of MOAS: memory object allocations, memory object operations, or the correlations between memory objects and input fields. Thus, MOAS comparison is able to interpret identified differences with the context information represented by MOAS. Compared to existing tools [27, 34, 49] that highlight differences in assembly instructions, the program representation by MOAS enables us to deliver comparison results more concisely.

Figure 2 illustrates the MOAS comparison result for our running example, which only has six misalignment places. Recall that BinDiff reports 30 misalignment differences. Figure 2(a1) and (b1) present the allocated memory objects, and Figure 2(a2) and (b2) show how these memory objects are referenced during runtime execution. For the second memory object access in Figure 2(a2) and (b2), we observe that their contexts, operations (call vsprintf), operation types (both are W), and correlated inputs (with the interval [0x4, 0x15d]) are identical. However, as shown in Figure 2(a1) and (b1), the allocations of L2 and R2 are different, as L2 is on the stack whereas R2 is on the heap. As L2 and R2 are different, the follow-on accesses to L2 and R2 will be different as well. Similarly, L3 and R3 are different, which leads to the difference between the accesses to L3 and R3. In addition to inferring that the patch fixes the vulnerability by updating data structures, Figure 2 also shows a telling clue that cannot be obtained from Figure 5.

Note that both R2 and R3 are two dynamically allocated memory objects in the heap to store the input. However, if we zoom in on their memory object representations shown in Figure 2(b1), we notice only R3’s malloc size is tainted—meaning that the size of R3 depends upon the input. This discrepancy motivates us to investigate why R2’s malloc size is not tainted. We perform backward slicing from R2’s malloc size and find that the source is a const string located in the read-only data section. That means the size of...
which enables us to recover function call stacks and excavate memory objects. To track dynamically allocated memory objects in heap and stack, we have to intercept related memory allocation system/library calls (e.g., `mmap`, `malloc`, and `alloca`).

4.2 Dynamic Taint and Execution Monitoring

The component of dynamic taint and execution monitoring in PatchScope is built on top of DECAF [28], which is a QEMU-based whole-system dynamic binary analysis platform. We adapt DECAF to our needs in supporting multi-tag taint propagation and recording all necessary runtime information used in the follow-up analysis.

Multi-tag taint analysis is indispensable to the correlations between memory objects and input fields. Our taint analysis begins with labeling each input byte as a unique taint tag. During taint propagation, if the multiple source operands of one instruction are tainted, we will set the taint tags of the destination operand as the union of all source operands’ taint tags.

DECAF also records fine-grained runtime information such as concrete execution states (e.g., instructions and operand values), which enables us to recover function call stacks and excavate memory objects. To track dynamically allocated memory objects in heap and stack, we have to intercept related memory allocation system/library calls (e.g., `mmap`, `malloc`, and `alloca`).

4.3 Function Call Stack Identification

The allocations and accesses to memory objects bind with their calling contexts. Therefore, our first step is to recover function call stacks. For tracking function call stack, the most straightforward way is to match the `call`/`ret` pairs and the balance of stack pointers. A main challenge is security patches may trigger some latent compiler optimization options that can mislead stack frame identification.

Common optimizations include tail call optimization [11] and function inline. The tail call optimization avoids the overhead of frequent stack frame set-up and tear-down. It switches to a `jmp` instruction at the end of the caller function to enter the callee function, instead of the general `call` instruction. GCC/Clang `-O2` and `-O3` enable this optimization. Appendix C shows a tail call optimization example. In Apache-1.3.35—one of our tested programs, the proportion of `jmp`-based calls encountered at run time is about 12%.

We leverage the recent progress in dynamic function call detection work, iCi [17], to detect `jmp`-based inter-procedural calls. iCi proposes a set of heuristics to filter out intra-procedural jumps (e.g., the jump target is within a function scope) and `jmp`-based calls (e.g., the jump target is a known function entry point). In this study, we perform the iCi analysis at the end of execution so that we have sufficient knowledge about function entry and exit information.

With the impact of function inline, there is no explicit control flow transition between an inline function and its caller. For this problem, we just treat them as an extended function frame. If the function inline only exists in one of the unpatched and patched programs, the contexts for allocating memory objects will be different. Then the MOAS comparison will identify more differences.

We leave such misleading differences to analysts for further inspection. It is practical and not difficult, because all elements except the contexts in two corresponding memory object accesses will be the same, if and only if the differences are caused by function inline.

4.4 Excavating Data Structures & Input Fields

All types of data structures in source code are compiled into memory objects in binary code. Our approach of excavating memory objects is inspired by previous work, Howard [57]. The basic insight is that memory access patterns reflect the types of program data structures. Therefore, we recover memory objects by tracing pointer propagations, as memory accesses in binary code are implemented via pointers, either using direct addressing or indirectly via registers. Our differential with Howard [57] is we need to establish the correlations between reverse-engineered memory objects and input fields. Following this requirement, our approach aims to identify only memory objects that are used to manipulate program inputs, instead of all the memory objects that are referenced during execution.

Root Pointer Extraction. The allocation of a memory object typically returns a pointer for further access and reference. We denote this pointer as a root pointer. In general, root pointers are unique for different memory objects, and they are not derived from any other root pointers. Following this definition, we identify memory objects by extracting their root pointers.

For the three types of memory objects (see Definition 1), extracting the root pointer for a static variable or a dynamic variable
in the heap is straightforward. We directly use a static variable’s memory address, which will not be reused, to represent its root pointer. For dynamic variables in the heap, we hook related memory allocation functions and take the return value as the root pointer. In Figure 4(b), eax holds the return value of malloc. Thus eax refers to the root pointer of the allocated memory object.

Identifying the root pointer for a local variable is a little trickier. Typically, a function stack frame holds the set of local variables, and programs address a local variable with the base pointer ebp. Thus, the root pointer of a local variable is indicated by the offset from ebp, such as ebp-0x4. For example, the first instruction (lea -0x172(%ebp),%edi) in Figure 4(a) indicates a root pointer with the address ebp-0x172, because it is in the current stack frame and is not derived from any other root pointers. The tricky issue here is a local variable can also be addressed via the stack pointer esp, which may lead to false positives on memory object identification. Our solution is to normalize these two possible local variable access ways only during execution enables us to examine whether two pointers are aliased. In Figure 4(a), the destination operand of mov $0x804b108,0x4(%esp) is a parameter of malloc. Function parameters are often aliased during execution enables us to examine whether two pointers are aliased. However, due to compiler optimization, the parameter here is directly addressed by esp. Specifically, the concrete execution state that is recorded during execution enables us to examine whether two pointers are aliased.

For a dynamically allocated memory object, the allocation function’s parameter explicitly indicates the size. The obstacle is to infer the size for local variables and static variables, because their types are invisible in binary code. We approximate their size by calculating the offset interval between two contiguous root pointers. For the example in Figure 4(a), the size of two memory objects (200) is just the offset interval between adjacent root pointers. The inferred size for a memory object may not be accurate, which could be larger than its definition if the adjacent memory object is not referenced during execution. However, we argue that the comparison between two memory object access sequences can counteract this inaccuracy.

Tracking Pointer Propagation. Through recognizing root pointers, we can identify the definition and allocation of memory objects. The next step is to extract the references to these memory objects, which is more challenging because programs generally define memory alias or transfer pointers to registers as de-references. Furthermore, without context information, the usage of a register is very similar when it is used as an address or a scalar. To solve this problem, we also adopt similar heuristics in Howard [57] by analyzing the context and tracking the movement of root pointers.

We first identify all root pointers. Second, we identify alias pointers by tracking the data movement of root pointers. Further, we track all arithmetic calculations on root pointers, because the memory address for an internal item of a data structure is typically calculated by adding an offset on the root pointer. Finally, for a memory load/store, we identify its root pointer by checking the data dependency of its memory address. Take the addressing mode of x86 instructions as an illustration. An address is computed as address = base + (index x scale) + offset, where base means the base address. By tracking the movement of root pointers, we can identify the pointer on which base depends as the root pointer.

Correlating Input Fields. We leverage multi-tag taint propagation results to establish the correlation between memory objects and input fields. For each tainted operand, we first determine its taint tags. If the operand is a register, we backtrack the execution trace to obtain the dependent memory operand. Second, we extract the root pointer, and the offset from the root pointer to the base address of the memory operand. Third, we construct the correlation between the referenced memory object and inputs with taint tags.

For an input field that contains multiple bytes such as a buffer or a string, a program generally allocates an array and manipulate them in a loop or C-string library function [57]. Within a loop, a program
typically manipulates consecutive input bytes via corresponding memory cells with the same root pointer and consecutive offsets. Based on the heuristics, we further group memory accesses that manipulate input fields (such as a string) via the same memory object (such as an array). Specifically, we first identify memory cells that belong to the same memory object. If two memory cells with consecutive offsets from the root pointer are used to manipulate consecutive input bytes, then we can group them into one memory object access.

4.5 MOAS Construction

After we excavated memory objects and correlated input fields, we have confirmed three items, \textit{mobj}, \textit{cc}, and \textit{a}, for a memory object access as Definition 2, leaving \textit{op} and \textit{optype} undecided.

For a data movement instruction, we track the memory load/store for its tainted operands. The \textit{optype} of \texttt{read} or \texttt{write} for the related memory object is decided by whether the tainted operand is loaded from or stored into memory.

Dealing with arithmetic instructions is a bit more complicated, because it is likely that only register operands are involved in an arithmetic instruction. We first identify whether a tainted register is a copy of a memory object. If yes, we take the arithmetic opcode as the \textit{op} and identify \textit{optype} by whether the tainted register is a source or destination operand.

For a C library call instruction that also propagates taint tags, we have optimized it using function summary at run time. Here we update the related memory object’s \textit{op} and \textit{optype} according to the function summary’s semantics.

Till now, we construct a memory object access with the root pointer of such memory object, the calling context, data-flow-related opcode, access type, and the correlated input field bytes that are addressed by the same root pointer. Then, we form a sequence with all the constructed memory object accesses in temporal order.

4.6 MOAS Alignment

To identify patch differences, we explore sequence alignment algorithms. The longest common subsequence (LCS) algorithm looks at the entire sequence, but it does not always deliver the most meaningful alignment in our context. In this study, we leverage the Smith-Waterman algorithm [58], which tends to find similar local regions between two sequences.

A challenge in MOAS alignment is that memory objects are represented as root pointers, which could be different for the same memory objects. To address the challenge, we treat a memory object access as a vector and measure two vectors’ similarity. In detail, both memory object and memory object access can be represented as vectors according to Definition 1 and Definition 2. For a pair of memory object accesses, we first calculate the similarity between two corresponding memory objects and then quantize the similarity between two memory object accesses. The details about how we apply the Smith-Waterman algorithm are presented in Appendix D.

5 EVALUATION

The prototype of PatchScope includes a total of 2,070 lines of C and Python code. 560 lines of C code is used to extend DECAF [28] for multi-tag taint analysis and execution monitoring. Our offline analysis, including memory object excavation and local sequence alignment, is implemented with 1,510 lines of Python code.

We conduct our experiments with three objectives. First, we did a comparative evaluation to demonstrate that PatchScope is particularly fit for capturing security patch differences. Second, to demonstrate that differences identified by PatchScope have rich semantics, we present the patch impacts on memory object accesses and input manipulations, by inspecting different items identified by PatchScope. Third, we perform case studies to show that PatchScope can assist reverse-engineers in further patch analysis.

5.1 Experiment Setup

Our experiment platform contains two Intel Xeon Gold 6134 processors, two GeForce GTX 1080 Ti 11GB graphic cards, and 256G memory, running Ubuntu 18.04 LTS.

Datasets and Ground Truth Collection. We select real-world vulnerabilities for evaluation considering several factors, including the vulnerability type, the patch complexity, and the patch patterns reported in Table 1.

Considering these factors, we select vulnerable applications from the datasets [37, 41, 46, 65, 68] according to the following criteria: 1) the vulnerable program, the security patch, and a PoC are all available; 2) types of vulnerabilities include stack overflow, heap overflow, integer-to-buffer overflow, off-by-one, use-after-free, and double-free; 3) patches cover all types listed in Table 1. Besides, we exclude vulnerable applications (such as 64-bits and interpreted language engines) that cannot be supported by our prototype. In this way, we select 37 applications in total from these datasets.

For each application, we use the same compiling options to generate two binaries, an unpatched one, and a patched one, respectively. To collect the ground truth, we manually identify patch differences between each pair of binaries with the assistance of patches.

Besides the 37 applications of which the patch source is available, we also select 8 applications of which the patches are not available. That is, the vulnerabilities in these applications are fixed in the new release versions instead of patches. This patching type represents a more challenging case for patch analysis, because the update version may contain security patches, general bug fixes, and functionality changes. As a result, it is difficult to identify and distinguish security patch differences. These 8 applications are listed in the last eight rows in Table 2. Through these applications, we aim to demonstrate that PatchScope can facilitate patch analysis with the assistance of rich information from identified differences. To collect ground truth for these eight applications, we manually identify their patched code in binaries by debugging PoCs.

Baseline Techniques. We select both prominent static and dynamic diffing approaches as baseline techniques.

We select three industry-standard binary diffing tools: BinDiff [27], Diaphora [34], and DarunGrim [49]. We also select an AI-powered binary diffing technique, DeepBinDiff [19], which is recently published and open-source. These four techniques work on static binaries directly. For a fair comparison, we setup dynamic execution comparisons by leveraging binary diffing techniques to compare execution traces caused by PoCs. For all different items identified by binary diffing, we regard an item as a difference only if it is
traversed in dynamic executions. With this setting, these four tools output two results for each application: all differences from the comparison between binaries (indicated by “static” in Table 1), and trace-relevant differences (indicated by “trace” in Table 1) from the comparison between executions caused by PoCs.

CoP [40] is a representative binary differencing technique, BinSim [44] and BLEX [20], as baseline techniques. BinSim [44] performs system call slice to segment equivalence checking to find relations between binaries. BLEX [20] collects dynamic behaviors such as memory accesses and syscalls during execution and then computes the similarity of the functions.

**Evaluation Metrics.** Since PatchScope and baseline techniques have metrics to represent differences, directly showing their results is not informative. As a normalization, we represent their results by counting the number of different units, including different instructions (BinDiff, Diaphora), basic blocks (DarunGrim, DeepBinDiff, and CoP), system calls (BinSim), memory cells (BLEX), and memory object access items (PatchScope).

### 5.2 Locating Patch Differences

Table 2 summarizes our comparative evaluation results in terms of located patch differences. The left three columns list vulnerable programs, CVE ID, and vulnerability types. The following two columns list lines-of-code changes and security patch types. For the last eight applications, as we do not have their concrete security patch information, the corresponding items (“LOC” & “Type”) are denoted as “NA”. The two sub-columns under “Time (s)” shows PatchScope’s overhead. The runtime overhead imposed by multi-tag taint analysis and execution monitoring (as shown in “Trace” sub-column) is about 10X slowdown on average. “Diff” sub-column shows the running time of PatchScope’s offline comparison. Considering that PatchScope attempts to free security professionals from the burden of manually reverse-engineering security patches, PatchScope’s overhead is acceptable.

By examining the numbers of located differences from Table 2, we have several observations. The most important take-away information is PatchScope delivers more accurate results than other techniques. PatchScope and DeepBinDiff identify much fewer differences than other techniques. Intuitively, this result indicates that PatchScope and DeepBinDiff are more effective. A deeper analysis shows that DeepBinDiff fails to identify more patch differences.
than PatchScope. We will compare their false positives and false negatives as follows.

Second, three industry binary diffing tools and BLEX [20] report more numbers than other techniques. By associating the numbers of differences and the lines-of-code changes, we further observe that these tools work well when the small LOC changes explicitly modify CFG/CG structures. However, with the LOC number increasing, they suffer from a large number of low-level assembly code differences.

Third, BinSim identifies patch differences accurately only in 12 applications. This result confirms our theoretical analysis that program representations used in semantics-aware binary diffing techniques are too coarse-grained to fit patch analysis, because many security patch types do not involve any API/syscall changes.

**Static vs. Dynamic.** Static and dynamic techniques have their own advantages. Dynamic techniques based on PoCs can filter out patch-irrelevant differences. However, dynamic techniques may also miss patch-relevant differences due to incomplete coverage.

By comparing the numbers of identified differences, we can further observe that dynamic techniques identify fewer differences than static techniques for most of the applications. For example, trace-based binary diffing reports much less number of differences than static-based binary diffing, except for 7 applications (including newspost-2.1, libbindfle-1.0.25, ringtonetools-2.22, O3read-0.03, conquest-8.2a, libsmi-0.4.8, and 2fax-3.04).

For these 7 exception applications, we manually examined the code changes and execution traces, and find that most of the identified differences are duplicated within a loop. That is, patches for these seven applications involve a code change with a loop. For example, the security patch in newspost-2.1 checks the index of an array within its boundary for each memory access to the internal bytes of the array. Then, the code change corresponding to the check will be repeated along with the execution.

To move further, we observe that PatchScope identifies fewer differences for these seven applications, which is less impacted by code changes in a loop. For example, trace-based binary diffing tools report thousands of differences in newspost-2.1, whereas PatchScope reduces the large difference number to only 1. To confirm this result, we inspect the result of PatchScope and observe that PatchScope does not capture the difference caused by the security check on index, because the index does not directly depend on program inputs and thus is not tainted. Instead, PatchScope detected a pair of different memory object access items, where the continuous bytes (the element \(a\) in Definition 2) differ. With this information, we can infer that the patched program regulates the handling of the PoC by cutting the received input. This case demonstrates that the memory object access enables PatchScope to capture higher-level patch differences beyond assembly code.

**False positives and False Negatives.** To verify that located differences are patch relevant, we manually examine identified differences, and calculate the false positives and false negatives based on the ground truth. Unlike binary diffing techniques that directly identified different instructions or blocks, BLEX [20] identifies different memory accesses regarding values of memory cells, and PatchScope identifies different memory object access items. To calculate false positives and false negatives, we associate the identified item with patched code via its operation instruction. The statistics are shown in Table 3.

The false-positive rate is calculated as the ratio of the number of patch-irrelevant differences to the number of all identified differences. Please note that we only calculate false positives for the 37 applications of which the patch source is available. Among these 37 applications, only one patch is involved for each pair of patched and unpatched applications. For the other 8 applications where the vulnerabilities are fixed in update versions, the updated versions may contain multiple fixes and there are multiple differences indeed. Thus, we cannot regard an identified difference that is irrelevant to the fixed vulnerability as a false positive.

A false negative refers to a missed patch difference. The false-negative rate is calculated as the ratio of the number of missed patches to the number of all patches. In total, the statistics of false positives covers 37 applications, and the statistics of false negatives covers all 45 applications.

As shown in Table 3, the false-positive rate in PatchScope is a little larger than that in the trace-based DeepBinDiff, but much smaller than those in other techniques. In fact, some patch-irrelevant MOA items identified by PatchScope are affected by the patch. For example, if a memory object is overwritten during the execution of a PoC, accessing this memory object will be identified as a difference by PatchScope. To alleviate false positives, we can further tie different MOA items according to their data dependencies. By contrast, false positives in other techniques lack such associations.

CoP [40] encounters more false negatives, because CoP does not deal with library functions. Thus, changes related to library functions will be missed by CoP. We observe that the false-positive rate of BLEX is pretty high. As BLEX [20] identifies different values read from or written to memory cells, code changes would result in a number of differences, including function pointers, instruction addresses, function parameters, and values of variables. By contrast, PatchScope outperforms BLEX [20] on two aspects: 1) PatchScope identifies differences on top of the reverse-engineered memory objects, which is a higher-level program abstraction than memory cells in BLEX [20]; 2) PatchScope only captures differences regarding input manipulations via memory objects, whereas BLEX [20] identifies all memory access behaviors.

We can observe that PatchScope encounters the least false negatives, which fails to identify patch differences for only one application. The only one false negative is caused by an off-by-one

<table>
<thead>
<tr>
<th></th>
<th>FPR</th>
<th>FNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinDiff</td>
<td>static</td>
<td>42.23%</td>
</tr>
<tr>
<td></td>
<td>trace</td>
<td>32.67%</td>
</tr>
<tr>
<td>Diaphora</td>
<td>static</td>
<td>39.29%</td>
</tr>
<tr>
<td></td>
<td>trace</td>
<td>29.37%</td>
</tr>
<tr>
<td>DarunGrim</td>
<td>static</td>
<td>33.75%</td>
</tr>
<tr>
<td></td>
<td>trace</td>
<td>22.76%</td>
</tr>
<tr>
<td>DeepBinDiff</td>
<td>static</td>
<td>18.91%</td>
</tr>
<tr>
<td></td>
<td>trace</td>
<td>12.12%</td>
</tr>
<tr>
<td>BLEX</td>
<td></td>
<td>81.56%</td>
</tr>
<tr>
<td>CoP</td>
<td></td>
<td>27.12%</td>
</tr>
<tr>
<td>BinSim</td>
<td></td>
<td>0.00%</td>
</tr>
<tr>
<td>PatchScope</td>
<td></td>
<td>14.73%</td>
</tr>
</tbody>
</table>
5.3 Interpreting Identified Differences

In terms of granularity, differences identified by PatchScope is the most fine-grained and informative one among these units. According to Definition 1 and Definition 2, different items in MOA include details of memory objects, contexts, operations on memory objects, and correlated input manipulations. Such contexts can interpret and determine differences caused by security patches.

To demonstrate that PatchScope delivers rich information for interpreting differences, we summarize the details of different MOA items identified by PatchScope, and present detailed elements between a pair of MOA items. Besides, we also present the impacts on input manipulations by examining the elements of $\alpha$. Please note that elements can impact others. For example, changes of $cc$ and $op$ typically lead to different $\alpha$. For this problem, we mark $\alpha$ as different if and only if the elements of $\alpha$ differ.

From Table 4, we observe that security patches include changes in both memory objects and the accesses to memory objects. For security patches that induce no changes in assembly code, such as streamripper and Fontforge, PatchScope can detect different MOA items via the impact on input manipulations ($\alpha$). Indicated by such information, a reverse-engineer can reason this difference by backtracking the contexts for calling library functions, and identify the different parameters.

On the contrary, PatchScope can also locate patch-relevant differences with our program abstraction for complicated patches.

### Table 4: Interpreting differences via the elements in MOA.

<table>
<thead>
<tr>
<th>Diff in MO</th>
<th>impacts on input manipulations</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc</td>
<td>cut the inputs</td>
</tr>
<tr>
<td>size</td>
<td>0</td>
</tr>
<tr>
<td>type</td>
<td>0</td>
</tr>
<tr>
<td>cc</td>
<td>0</td>
</tr>
<tr>
<td>op</td>
<td>7</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>10</td>
</tr>
</tbody>
</table>

For example, the patch for 2fax-3.04 overwrites a function. Most of the baseline techniques identify a pair of unmatched functions and a large number of unmatched items, which is impractical to interpret these differences as security experts could be plagued by such a large number of low-level code differences. With the assistance of PatchScope, we find that the MOAS from the patched program adds a security check. More details are presented in Appendix C.

Another remarkable case is putty, of which the patch is unavailable. Its new version contains quite a lot of updates. Diaphora shows up to 696 different instructions. BLEX [20] identifies up to 4564 different memory accesses, and these differences would overwhelm reverse-engineers. Besides, it is difficult to extract semantics and contexts from these differences.

To further reveal the advantages that PatchScope provides rich patch context information that other tools cannot offer, we present several case studies in the appendix to show how PatchScope can assist patch analysis.

6 DISCUSSION AND CONCLUSION

Our approach relies on dynamic taint analysis to excavate program data structures. It may lead to under-tainting problems caused by control-flow dependencies and pointer tainting. A possible solution, like DTA++ [32] and pointer tainting policy [56], is to add additional taint rules for implicit data flows.

We focus on security patches for memory corruption. Our insight that most security patches aim to better regulate the handling of bad inputs may also apply to other types of vulnerabilities such as permission bypassing. We will leave it as future work.

The evaluation only consists of x86 applications, because the runtime analysis is built on top of DECAF [28] and DECAF for supporting x86-64 is still under development. To support x86-64 applications, we should revise our algorithm for excavating program data structures as the calling convention changes. We believe it is not difficult since the calling convention in x86-64 should also follow memory access patterns.

Patch analysis is a prominent application of binary diffing techniques. In this paper, we develop a memory object centric dynamic approach for patch analysis. Our technique can not only capture small security patch differences but also reveal more patch details. Security professionals utilizing PatchScope will enjoy a simpler and more streamlined patch analysis process than ever before.

ACKNOWLEDGMENT

This work is partly supported by National Natural Science Foundation of China under Grant No.61672394, U1836112, and 61876134. Jiang Ming and Haotian Zhang were supported by National Science Foundation (NSF) under grant CNS-1850434.

Jeong Wook Oh. 2009. Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries. Black Hat USA.


APPENDIX

A OUTPUT RESULTS FROM BINDIFF

For the example in Figure 1, we leverage BinDiff to compare the dynamic execution traces of the two httpd versions on a given PoC. It reports up to 30 differences in terms of the instructions removed or added. We list a snippet of the instruction alignment sequence in Figure 5.

<table>
<thead>
<tr>
<th>ghttpd-1.4.3</th>
<th>ghttpd-1.4.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor %eax,%eax xor %eax,%eax</td>
<td></td>
</tr>
<tr>
<td>lea 0x374(%esp),%eax</td>
<td>mov %bx,0x3b(%esp)</td>
</tr>
<tr>
<td>mov $0x2e64,0x27(%esp)</td>
<td></td>
</tr>
<tr>
<td>lea 0xbd(%esp),%esi</td>
<td></td>
</tr>
<tr>
<td>mov %bx,0xc3b(%esp) mov %edi,(%esp)</td>
<td></td>
</tr>
<tr>
<td>mov %eax,0xc(%esp) mov %edi,0x10(%esp)</td>
<td></td>
</tr>
<tr>
<td>mov %eax,0x10(%esp)</td>
<td></td>
</tr>
<tr>
<td>lea 0x374(%esp),%eax</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: A partial sequence of Figure 1’s instruction alignment returned by BinDiff.

With the output like Figure 5, it is far from meeting the goal of patch analysis. First, we can observe that BinDiff did an inaccurate trace alignment. The instruction at Line 19 in the left should be aligned to Line 4 in the right. Second, an expert may infer that it is an out-of-boundary vulnerability, because the patch redefines two dynamically allocated heap spaces with flexible length. Even so, low-level differences cannot avail security experts much regarding how to trigger this vulnerability.

B OUTPUT RESULTS FROM BLEX

For the example in Figure 1, BLEX [20] can successfully identify the two different functions because the similarity score is pretty low. However, if we employ BLEX [20] to identify differences for patch biffing, BLEX [20] reports 32 different memory accesses, which are shown in Figure 6.

Please note these differences only include memory accesses generated by the instructions belonging to the vulnerable function Log in Figure 1. That is, we exclude different memory accesses during the execution of library function calls. Otherwise, BLEX [20] would report thousands of different memory accesses, because this patch changes two local variables in the stack to dynamically allocated variables in the heap. As a consequence, all memory accesses during the library function calls (such as vsprintf and strlen) would be identified as differences, because of the memory cell types (stack vs. heap).

Our further analysis on the reported 32 different memory accesses shows that these differences include memory accesses via function pointers, instruction addresses, parameters, and so on. This result indicates that the patch in Figure 1 results in types of different elements regarding memory accesses. By contrast, PATCHSCOPE outperforms BLEX [20] as the memory object access is constructed on top of reverse-engineered memory objects and it only captures memory objects used for manipulating program inputs.

Through this example, we highlight code representations defined in binary diffing techniques are specific for problem scopes. As demonstrated in the literature [20], BLEX explicitly considers the case where different compilers and optimization settings produce different binary programs from identical source code. For patch analysis, however, BLEX may not be suitable because it reports a number of differences.

C TAIL CALL OPTIMIZATION EXAMPLE

Tail call optimization uses the stack memory more efficiently but hides inter-procedural call relationship. We simplify the example in Figure 7 with pseudo-assembly languages.
D SMITH-WATERMAN ALGORITHM

The input to Smith-Waterman algorithm is two sequences \( A = a_1, a_2, \ldots, a_n \) and \( B = b_1, b_2, \ldots, b_m \) of length \( n \) and \( m \) respectively. A maximum similarity matrix \( H \) is filled using the equation below.

\[
H(i,0) = 0, \quad 0 \leq i \leq m, \\
H(0,j) = 0, \quad 0 \leq j \leq n,
\]
and

\[
H(i,j) = \max \begin{cases} 0 \\ H(i-1,j-1) + \text{Sim}(a_i, b_j) \\ \max_{k \geq 1} H(i,j-k) + W_k \\ \max_{l \geq 1} H(i-1,j-l) + W_l \end{cases}
\]  \quad (1)

1 \leq i \leq m, 1 \leq j \leq n

where \( a_i \) and \( b_j \) are two memory object access sequences. \( \text{Sim}(a_i, b_j) \) is the similarity score of two memory object access items. We treat a memory object access’s definition as a vector and measure two vectors’s similarity using Jaccard index: \( |A \cap B|/|A \cup B| \). Both \( W_k \) and \( W_l \) are the gap penalty scheme. In our comparison, the simple gap penalty scheme that uses a fixed score for each gap already delivers good precision. We tune the value of \( W_k \) and \( W_l \) as -2.

E CASE STUDIES

Nginx-1.4.0. The vulnerability CVE-2013-2028 is a classic integer-to-buffer-overflow. A signed and negative integer is mistakenly used as a parameter for calling \( recv \). By comparing MOAS between two executions, we find that the MOAS from the patched program is much shorter than that from the unpatched one. With this observation, an intuitive inference is that the patched program blocks the bad inputs. Further, we identified an MOA item that only exists in the patched program, of which the operation code is \( memcpy \). This item confirms our inference that the patch fixes the vulnerability via a security check. This patch seems very simple. However, the long MOAS from the unpatched program indicates that the patch point is far away from the crash point, which motivated us to look for a deeper understanding of the fixed vulnerability. Finally, we identified a memory object access that was caused by a buffer overflow.

An interesting observation from this case is that the operations and corresponding variables in the code changes of patches may not be the vulnerable variables. For this example, it first receives an integer, and then uses this integer to allocate a second memory object and calculates the length for further received inputs. The mistake calculation leads to a fault parameter for invoking \( recv \). This observation also suggests that previous techniques [68, 74] that extract patch signatures, vulnerable operations, and vulnerable variables from patches should consider more patch patterns.

2fax-3.04. The patch for this application overwrites a function, which is the most complicated one. As shown in Table 1, the LOC is 27, and the industry binary diffing tools report a large number of differences. By examining these results, we observe that these industry binary diffing tools identified a pair of unmatched functions and a large number of unmatched items in the pair of functions.

Security experts could be plagued by such a large number of low-level code differences. By contrast, PatchScope only identified 4 differences. Examining the different details in MOA, we found that the new function in the patch only adds a security check. With the impact of the security check, the patched program blocks the PoC. This case further demonstrates that the core of PatchScope, memory object access sequences, is particularly fit for patch analysis by identifying differences in input manipulations.

Besides, we can also observe that DeepBinDiff [19] performs well in this case, as it identifies only three different basic blocks. To confirm this result, we discussed this case with the author of DeepBinDiff. The reason is DeepBinDiff adopts a program-wide code representation learning, whereas traditional binary diffing works on function similarity.

libsndfile. This vulnerability is difficult to understand due to complicated program logical faults. The patch for fixing this vulnerability changes multiple statements in different locations. Although complicated, PatchScope still provides valuable clues to reason about the vulnerability. First, PatchScope successfully narrows down the patch differences to only three different memory object access items, which are marked with colors in Figure 8.

We can observe one extra sub manipulation on memory object \( R_1 \) and \( R_3 \) in the patched program, and this manipulation calculates the value of \( R_3 \). Thus, \( R_3 \) is passed to \( memcpy \) as a parameter. By contrast, the vulnerable program takes \( L_2 \) as the parameter of \( memcpy \). We can learn that this patch fixes a vulnerability by changing the parameter value of \( memcpy \). As both \( L_2 \) and \( R_3 \)’s types are registers, we infer that both \( L_2 \) and \( R_3 \) are \( memcpy \)’s size parameter—number of bytes to copy. This finding motivates us to backtrack \( L_2 \) and \( R_3 \’s data flow in MOAS. We show their dependencies at the bottom of Figure 8.

The missing line of \( L_2 \)’s data flow dependencies reveals the root cause of the vulnerability: \( L_2 \) can be manipulated by syscall \( read \) before it is passed to \( memcpy \). The patch adds one line to initialize \( R_3 \) again before \( memcpy \) accepts it. Note that \( L_2 \) has two correlated input fields, and one of them [0x176, 0x303] does not exist in \( R_3 \). This clue implies that overflows this particular input field can trigger the vulnerability, and we confirm this in our PoC exploit.
Figure 8: PatchScope’s comparison result for libsndfile.