PatchScope: Memory Object Centric Patch Diffing

Lei Zhao[*†]
leizhao@whu.edu.cn
Wuhan University

Yuncong Zhu
yuncongzhu@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**.

KEYWORDS

Software security; vulnerability analysis; patch diffing

ACM Reference Format:
Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. 2020. PatchScope: Memory Object Centric Patch Diffing. In *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3372297.3423342

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

[*](1) School of Cyber Science and Engineering, Wuhan University, Wuhan, China; (2) Key Laboratory of Aerospace Information Security and Trust Computing, China
[†]Corresponding authors: leizhao@whu.edu.cn and jiang.ming@uta.edu

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3423342

flow/call graph [7, 25, 29]) to dynamic behaviors (e.g., system call dependence [5, 64]), and then leverage such code representations to measure the similarity between two binaries. The main limitation of these techniques is that code similarity representations are often specifically designed for different problem scopes, such as vulnerability detection [16, 18, 21, 22, 71], software plagiarism detection [40, 60, 64], or obfuscation-resilient malware comparison [5, 12, 44, 70]. As a result, these techniques may not be suitable to locate patch-relevant code for sophisticated types of patches. For example, up to 50% of security patches only modify less than 7 LOC [36]. Such a small degree of change includes resizing a buffer, changing function parameters, etc., which may not exhibit any code differences in generated binaries. Besides, false positives in binary diffing [19] also affect the performance of patch diffing.

Locating patch-relevant code changes is just an intermediate step. A more challenging task in patch analysis is to interpret identified differences for investigating the root causes of the fixed vulnerability, as well as for understanding patch details such as how the patch fixes the repaired vulnerability.

To interpret patch differences, most of the existing static techniques leverage security patch patterns. APEG [8] identifies a security patch if the corresponding code difference involves a security check. SPAIN [72] identifies security patches based on the assumption that a security patch will not change program semantics and it typically introduces a new branch for detecting invalid inputs. SID [68] determines security impacts based on security patterns by summarizing patterns of how security operations fix common types of vulnerabilities.

However, security patch patterns used in existing techniques [8, 68, 72] may be incomplete to characterize types of patches. From the perspective of programming, a vulnerability is mostly caused by a specific programming logical fault. For example, a buffer overflow can be caused by missing the input check, or error parameters for invoking library functions. To fix the vulnerability, a patch will consequently depend on specific program logics, which would vary from vulnerability to vulnerability. To fix a use-after-free or double-free vulnerability, a typical pattern is to make the invalid pointer *NULL*. Another common pattern is to delete the dereference. With the impact of sophisticated types of patches, security patch patterns may be incomplete to interpret patch differences.

Taking advantage of detailed run-time information, execution comparison techniques [53, 75] have also been proposed for reasoning execution differences. For an observed crash, differential slicing [30, 67] constructs relevant control and data dependencies for causal analysis. Although dynamic techniques often assume a PoC is available, investigating the root cause of a vulnerability and understanding patch details is still difficult, which may take a significant amount of time and domain knowledge [6]. Moreover, identified differences at the binary level are represented as low-level instructions and operands. Without the support of high-level program abstractions, low-level differences fail to deliver rich semantics for understanding how or why the two executions differ.

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 [9]. At the binary level, a data structure reference is represented as a *memory object access*. Instead of syntactical code changes, our insight regarding how a security patch affects program semantics is: *it typically modifies the manipulations of 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.

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 fields. In this way, we abstract the semantics of a dynamic execution as a memory object access sequence. At last, we adopt a local sequence alignment algorithm from bioinformatics [58] to identify patch differences.

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.

Table 1: Statistics on different memory corruption patch types from five recent datasets [37, 41, 46, 65, 68]. The number of LOC changes ranges from 1 to 43, and 5 is the median number.

No.	Category	Percentage
1	add input sanitization checks	43.5%
2	change input sanitization checks	25.1%
3	add data structures	6.1%
4	change data structure definitions	6.5%
5	change data structure references	22.3%
6	change function parameters	10.9%
7	add or change function calls	15.3%
8	add functions	4.7%
9	change functions	7.6%

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.

1	int **main**() { …; **serveconnection**(sockfd); …; }	
2	int **serveconnection**(int sockfd) {	
3	char tempstring[8192]; // *tempstring* is allocated to store program inputs.	
4	**Log**("Connection from %s, request = \"GET %s\"", inet_ntoa(sa.sin_addr), ptr); // *ptr* is a pointer to *tempstring* …; }	
5	void **Log**(char *format, …){	void **Log**(char *format, …){
6	char temp[200], temp2[200];	char *temp, *temp2;
7		temp=malloc(strlen(format));
8	vsprintf(temp, format, ap[1]);	vsprintf(temp, format, ap[1]);
9		temp2=malloc((strlen(temp)+strlen(datetime_final)+5));
10	sprintf(temp2, "%s - %s\n", datetime_final, temp);	sprintf(temp2, "%s - %s\n", datetime_final, temp);

(a) ghttpd-1.4.3 (b) ghttpd-1.4.4

[1] ap in Line 8 is the type of *va_list*, which stores a variable arguments list from program inputs.

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 *%39s*. 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 irrelative 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 calls, 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*.

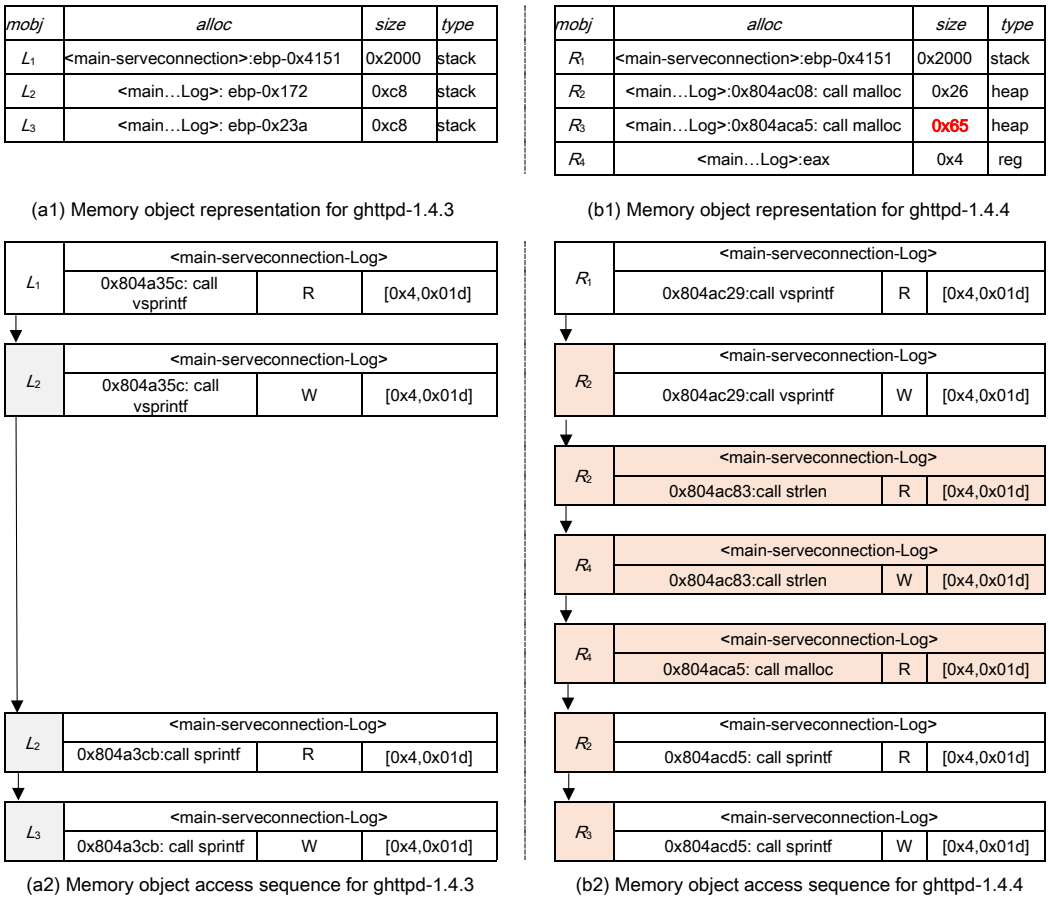
We illustrate *alloc* using our running example from Figure 1. The function *Log* defines a local variable *temp* with 200-bytes length. For the dynamic execution on ghttpd-1.4.3, the *alloc* for *temp* will include the calling context, *main-serveconnection-Log*, and the offset from the frame pointer 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_1" (mobj) represents a memory object defined in (a1); "<main-serveconnection-Log >" (cc) is the function that references "L_1"; "0x804a35c:call vsprintf" (op) refers to the instruction operating on "L_1"; "R" (optype) means the access type of read; "[0x4,0x01d]" (α) indicates the input field that is correlated with "L_1".

Figure 2: MOAS comparison result for the patch differences shown in Figure 1. We highlight the misalignment places. "0x65" in b_1 indicates that the value of *size* is tainted.

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 L_2 and R_2 are different, as L_2 is on the stack whereas R_2 on the heap. As L_2 and R_2 are different, the follow-on accesses to L_2 and R_2 will be different as well. Similarly, L_3 and R_3 are different, which leads to the difference between the accesses to L_3 and R_3. 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 R_2 and R_3 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 R_3's *malloc* size is tainted—that means the size of R_3 depends upon the input. This discrepancy motivates us to investigate why R_2's *malloc* size is not tainted. We perform backward slicing from R_2's *malloc* size and find that the source is a const string located in the read-only data section. That means the size of

R_2 is fixed and decided by this const string. We have verified that an unsafe input can still overflow R_2 to crash ghttpd-1.4.4 if the number of input bytes exceeds the const string length. Our CVE ID request for this "buggy patch" is under review.

4 PATCHSCOPE SYSTEM DESIGN

The section presents our design details to unleash the power of MOAS comparison for security patch analysis.

4.1 System Overview

Figure 3 shows the overview of PatchScope. It contains four main components: dynamic taint and execution monitoring, memory object excavation, memory object access construction, and MOAS alignment.

First, we leverage a fine-grained dynamic tainting technique to monitor dynamic executions for both P and P' on the same PoC. The fine-grained dynamic tainting component gives each input byte a unique taint tag, and records the propagations of all tainted bytes. Meanwhile, the execution trace records all executed instructions, all operand values as well as their taint tags.

Second, we dynamically excavate memory objects from execution traces. The basic insight is that memory access patterns reflect the types of program data structures. Moreover, our approach establishes the correlations between reverse-engineered memory objects and input fields, for identifying memory objects that are used to manipulate program inputs.

Third, we construct memory object access sequences for both P and P', according to **Definition 1** and **Definition 2**.

Finally, we leverage a sequence alignment algorithm to identify differences by comparing memory object access sequences for both P and P'. To reason differences, we further use semantics reflected on MOAS to identify vulnerable program data structures, invalid inputs to trigger the vulnerability, as well as the context.

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 inlined 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

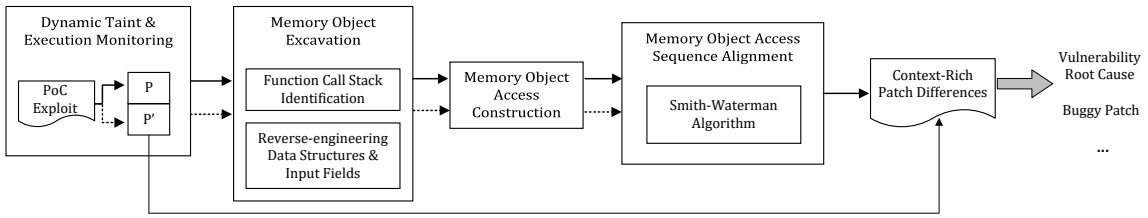


Figure 3: PatchScope system overview.

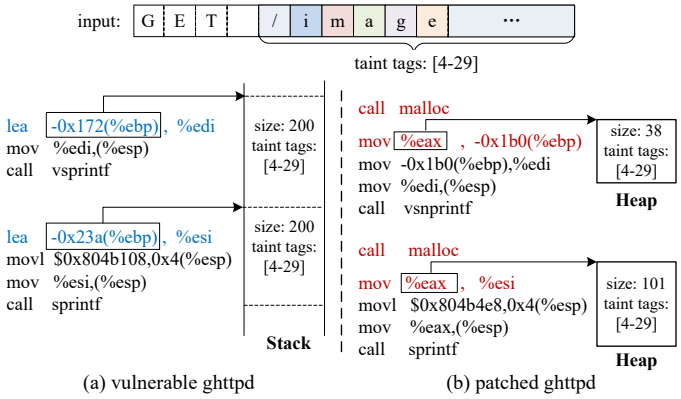


Figure 4: Reverse-engineered memory objects and their correlated input fields for the vulnerable and patched ghttpd.

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 via *ebp*. Specifically, the concrete execution state that is recorded 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 *sprintf*. Function parameters are often pushed into stack first. However, due to compiler optimization, the parameter here is directly addressed by *0x4(%esp)*. After our normalization, we regard *esp+0x4* as *ebp-0x24c* by calculating the offset of *0x4(%esp)* from *ebp*.

An exception happens when the compiler optimization option "- fomit-frame-pointer" is enabled, in which the generated binary code will not keep the frame pointer *ebp* for a function call. At the beginning of the function, there are no stack balance instructions such as *push ebp; mov esp, ebp*. This optimization avoids the instructions to save, set up, and restore frame pointers, but it complicates our root pointer identification for local variables. To address this problem, we represent the stack frame pointer with *esp*'s offset if we detect the non-existence of stack balance instructions.

Memory Object Size Inference 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 × 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, *mobj*, *cc*, and α, for a memory object access as **Definition 2**, leaving *op* and *optype* undecided.

For a data movement instruction, we track the memory load/store for its tainted operands. The *optype* of *read* or *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 *op* and identify *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 *op* and *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

Table 2: Comparative evaluation results. "OF" is short for "overflow".

Program	Vulnerability		Security Patch		Time(s)		Result (Number of different items detected)											
	CVE	Type	LOC	Type	Trace	Diff	BinDiff (Inst.)		Diaphora (Inst.)		DarunGrim (Basic Block)		DeepBinDiff (Basic Block)		BLEX (Inst.)	CoP (BB.)	BinSim (Syscall)	PatchScope (MOA)
							static	trace	static	trace	static	trace	static	trace				
streamripper-1.61.25	2006-3124	stack OF	6	data struc. & func. PRM	147	123	0	0	0	0	0	0	0	0	361	0	0	1
newspost-2.1	2005-0101	stack OF	2	change checks	141	92	4	2050	8	2052	2	1025	1	1024	2075	2	0	1
mcrypt-2.5.8	2012-4409	stack OF	2	add checks	99	81	15	14	45	38	6	5	4	2	151	3	0	3
tiffsplit-3.8.2	2006-2656	stack OF	5	function calls	108	88	5	5	5	5	3	3	2	2	29	2	4	4
unrar-3.9.3	NA	stack OF	4	function calls	291	135	11	6	11	6	2	1	3	1	41	2	5	3
xmp-2.5.1	2007-6731	stack OF	3	data structures	138	84	1	1	1	1	1	1	1	1	16	1	0	1
gif2png-2.5.2	2009-5018	stack OF	6	add checks	117	104	25	17	131	38	16	11	2	2	203	12	0	5
libsndfile-1.0.25	2015-7805	heap OF	5	function PRM	110	116	33	1579	97	2523	9	850	1	1	94	3	0	3
nasm-0.98.38	2004-1287	stack OF	2	function calls	222	148	3	2	2	2	1	1	4	4	43	1	3	2
0verkill-0.16	2006-2971	numeric error	1	change checks	129	91	2	2	2	2	1	1	0	0	87	1	0	2
ringtonetools-2.22	2004-1292	stack OF	4	data struc. & add checks	179	111	2	4082	2	4082	1	2041	1	2041	2359	1	0	2
Unalz-0.52	2005-3862	stack OF	2	add checks	81	104	22	22	16	16	4	4	2	2	245	3	0	2
O3read-0.03	2004-1288	stack OF	1	add checks	79	99	13	11264	5	5120	4	3072	1	1024	3323	2	0	1
gzip-1.2.4	2001-1228	stack OF	2	function calls	135	151	2	1	2	1	2	1	3	1	601	2	0	2
binutils-2.12	2005-4807	stack OF	8	function calls	103	138	7	2	4	2	4	1	2	1	92	2	2	2
conquest-8.2a	2007-1371	stack OF	5	change checks	1537	189	10	3072	36	5124	11	2049	0	0	1139	8	0	1
poppler-0.24.1	2013-4473	stack OF	4	function calls	444	163	2	2	2	2	1	1	0	0	121	1	0	3
ntpd-4.2.6	2014-9295	stack OF	11	add functions	208	157	52	45	70	55	15	11	5	5	76	7	0	6
libsmi-0.4.8	2010-2891	stack OF	7	change checks	178	125	37	2948	59	3092	8	259	4	258	1112	6	0	5
fontforge-20100501	2010-4259	stack OF	18	function PRM	277	148	0	0	0	0	0	0	0	0	72	0	0	4
2fax-3.04	2004-1255	stack OF	27	add func. & func. PRM	119	135	97	4588	100	5394	12	1080	3	768	1584	14	0	4
libpng-1.2.5	2004-0597	stack OF	25	add checks	69	110	21	8	47	12	13	5	0	0	111	11	0	5
ytree-1.9.4	NA	stack OF	4	function calls	52	82	9	9	9	9	1	1	0	0	81	1	0	3
sox-12.17.4	2004-0557	stack OF	8	add checks	101	90	76	27	26	13	8	3	6	3	372	5	0	4
ettercap-0.7.5.1	2013-0722	stack OF	2	function PRM	405	106	0	0	0	0	0	0	0	0	128	0	0	1
binutils-2.15	2006-2362	stack OF	15	add checks	157	143	107	32	189	28	35	10	8	2	337	25	0	5
prozilla-1.3.6	2004-1120	stack OF	10	function calls	189	130	11	2	21	6	6	1	6	1	436	6	0	3
nginx-1.4.0	2013-2028	numeric error	4	add checks	88	97	49	12	9	4	3	2	3	2	210	3	0	6
proftpd-1.3.3a	2010-4221	stack OF	4	add checks	111	91	5	5	3	3	1	1	1	1	189	1	0	2
tiff2pdf-4.0.9	2018-15209	heap OF	8	add checks	116	91	111	47	202	52	40	8	10	4	527	28	0	5
nginx-1.12.0	2017-7529	numeric error	3	add checks	309	173	15	8	16	10	6	2	3	1	532	4	0	3
Aircrack-ng-1.2	2014-8322	stack OF	2	add checks	455	187	6	3	34	2	2	1	2	1	324	2	0	3
leptonica-1.70.1	2018-7186	stack OF	2	function PRM	97	131	0	0	0	0	0	0	0	0	319	0	0	1
openjpeg-2.1.1	2016-7445	Null pointer	4	add checks	163	149	48	11	24	3	9	2	10	8	362	8	0	4
Jasper-1.900.9	2016-8887	Null pointer	5	function calls	193	127	32	23	24	15	5	4	4	4	144	4	0	3
libzip-1.2.0	2017-12858	double free	3	function calls	125	113	40	16	108	15	12	6	8	5	514	8	1	1
GraphicsMagick-1.3.26	2017-14103	UAF	51	function calls	643	190	275	35	361	49	50	10	19	10	634	41	11	5
putty-0.66	2016-2563	stack OF	NA	NA	339	176	675	63	696	21	110	5	33	9	4564	87	18	12
mutt-1.4.2.2	2007-2683	stack OF	NA	NA	312	131	31	2	33	4	8	1	9	2	148	4	6	2
inetutils-1.8	2011-4862	stack OF	NA	NA	137	159	2483	436	2506	597	441	155	207	81	6029	135	41	17
Apache-1.3.35	2006-3747	off-by-one	NA	NA	358	168	198	0	198	0	21	0	45	0	158	16	38	0
xrdp-0.4.1	2008-5904	heap OF	NA	NA	284	141	7	7	9	8	2	2	2	2	342	2	2	5
alsaplayer-0.99.80-rc2	2007-5301	stack OF	NA	NA	252	111	16	2	16	2	1	1	0	0	147	1	0	2
opendchub	2010-1147	stack OF	NA	NA	203	126	56	8	73	25	17	4	27	9	266	14	18	2
nasm-2.14	2018-19214	heap OF	NA	NA	209	125	1144	394	1409	493	166	46	33	24	3672	63	27	13
In Total					**10209**	**5729**	**5758**	**30852**	**6611**	**28926**	**1060**	**10687**	**475**	**5306**	**34370**	**542**	**176**	**164**

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 diffing tool based on symbolic execution. CoP identifies an equivalent basic block pair with symbolic execution and theorem proving and then measures the similarity with semantically equivalent basic blocks.

Besides, we select two dynamic and semantics-aware binary diffing technique, BinSim [44] and BLEX [20], as baseline techniques. BinSim [44] performs system call sliced 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*, *libsndfle-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 α 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.

Table 3: False positive and false negative rates.

		FPR	FNR
BinDiff	static	42.23%	15.56% (7/45)
	trace	32.67%	15.56% (7/45)
Diaphora	static	39.29%	15.56% (7/45)
	trace	29.37%	15.56% (7/45)
DarunGrim	static	33.75%	15.56% (7/45)
	trace	22.76%	15.56% (7/45)
DeepBinDiff	static	18.31%	26.67% (12/45)
	trace	12.12%	26.67% (12/45)
BLEX		81.56%	2.22% (1/45)
CoP		27.12%	51.11% (23/45)
BinSim		0.00%	73.33% (33/45)
PatchScope		14.73%	2.22% (1/45)

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

vulnerability in *Apache-1.3.35* and its patch. With basic code comparison, we identify that patch changes an instruction from *cmp* $0x5,%ecx to *cmp* $0x4,%ecx. All the techniques fail to identify this small code change. Through dynamic debugging, we observe that the off-by-one operation overruns a memory object with a pointer pointing to a memory object for received inputs. As our dynamic taint component only tracks input propagations on memory objects, pointers to these memory objects will not be tainted. Thus, our approach fails to identify this patch points.

PatchScope and other dynamic techniques may miss a part of patch-relevant differences if the corresponding code is not traversed. By examining the ground truth, we find that the vulnerability (CVE-2005-4807) exists in 4 different locations in *binutils-2.12*. The security patch fixes 4 different vulnerable statements. PatchScope only identifies 2 locations that are covered by the execution trace, because the PoC did not trigger the vulnerability at the other two locations. An interesting observation is that DeepBinDiff identifies only 2 differences, even though all the 4 differences are induced by the same fix.

By comparing the false negatives of PatchScope with other techniques, we find 4 applications of which the security patch differences can only be captured by PatchScope. Further inspection shows that these patches include resizing a buffer length, changing a parameter, and changing the value of a const variable. For example, the security patch of *streamripper* increases a buffer size from $0x32$ to $0xfb8$. The security patch of *Fontforge* changes a format string from %[\"] to %99[\"]. Other techniques fail to identify them because these patches induce no changes in assembly code. By contrast, PatchScope can identify them via MOAS comparison because such code changes induce different input manipulations.

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 MOAS 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 α. Please note that elements can impact others. For example, changes of *cc* and *op* typically lead to different α. For this problem, we mark α as different if and only if the elements of α 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 (α). 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.

		Impacts on input manipulations		
		cut the inputs	filter the inputs	None
Diff in MO	*alloc*	0	0	1
	size	3	0	0
	type	1	0	0
Diff in MOA	*cc*	0	17	0
	op	7	0	5
	α	10	0	0

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 a 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.

REFERENCES

[1] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. 2005. OPUS: Online Patches and Updates for Security. In *Proceedings of the 14th USENIX Security Symposium.*

[2] Frederico Araujo, Kevin W. Hamlen, Sebastian Biedermann, and Stefan Katzenbeisser. 2014. From Patches to Honey-Patches: Lightweight Attacker Misdirection, Deception, and Disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14).*

[3] Ionut Arghire. 2019. Patches for Internet Explorer Zero-Day Causing Problems for Many Users. https://www.securityweek.com/patches-internet-explorer-zero-day-causing-problems-many-users.

[4] Jeffrey Avery and Eugene H. Spafford. 2017. Ghost Patches: Fake Patches for Fake Vulnerabilities. In *Proceedings of the 32nd International Conference on ICT Systems Security and Privacy Protection (IFIP SEC'17).*

[5] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09).*

[6] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *Proceedings of 29th USENIX Security Symposium (USENIX Security'20).*

[7] Martial Bourquin, Andy King, and Edward Robbins. 2013. BinSlayer: Accurate Comparison of Binary Executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW'13).*

[8] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P'08).*

[9] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07).*

[10] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2095–2108.

[11] William D. Clinger. 1998. Proper Tail Recursion and Space Efficiency. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98).*

[12] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. 2010. Identifying Dormant Functionality in Malware Programs. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P'10).*

[13] Bart Coppens, Bjorn De Sutter, and Koen De Bosschere. 2013. Protecting Your Software Updates. *IEEE Security & Privacy* 11, 2 (March 2013), 47–54.

[14] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. 2007. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07).*

[15] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16).*

[16] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).*

[17] Franck de Goër, Sanjay Rawat, Dennis Andriesse, Herbert Bos, and Roland Groz. 2018. Now You See Me: Real-time Dynamic Function Call Detection. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18).*

[18] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19).*

[19] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DEEPBINDIFF: Learning Program-Wide Code Representations for Binary Diffing. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20).*

[20] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14).*

[21] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS'16).*

[22] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16).*

[23] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. 2010. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P'10).*

[24] Ulf Frisk. 2018. Total Meltdown? http://blog.frizk.net/2018/03/total-meltdown.html.

[25] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Poceedings of the 10th International Conference on Information and Communications Security (ICICS'08).*

[26] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, Heyuan Shi, and Jiaguang Sun. 2018. VulSeeker-pro: Enhanced Semantic Learning Based Binary Vulnerability Seeker with Emulation. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18).*

[27] Google LLC. 2011. BinDiff: Graph Comparison for Binary Files. https://www.zynamics.com/bindiff.html.

[28] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make It Work, Make It Right, Make It Fast: Building a Platform-Neutral Whole-System Dynamic Binary Analysis Platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14).*

[29] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. 2009. Large-scale Malware Indexing Using Function-call Graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09).*

[30] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. 2011. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P'11).*

[31] Mateusz Jurczyk. 2017. Using Binary Diffing to Discover Windows Kernel Memory Disclosure Bugs. Google Project Zero Team Blog.

[32] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11).*

[33] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. 2009. Effective and Efficient Malware Detection at the End Host. In *Proceedings of the 18th Conference on USENIX Security Symposium (USENIX Security'09).*

[34] Joxean Koret. 2015. Diaphora: A Free and Open Source Program Diffing Tool. http://diaphora.re/.

[35] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification.* Springer, 712–717.

[36] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17).*

[37] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18).*

[38] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-version Binary Code Similarity Detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18).*

[39] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. 2014. Sound Input Filter Generation for Integer Overflow Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14).*

[40] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14).*

[41] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP). IEEE.*

[42] Niccolò Marastoni, Roberto Giacobazzi, and Mila Dalla Preda. 2018. A Deep Learning Approach to Program Similarity. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis (MASES'18).*

[43] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Proceedings of the 16th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'19).*

[44] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17).*

[45] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2018. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering* 23, 5 (2018), 2901–2947.

[46] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the Reproducibility of Crowd-reported Security Vulnerabilities. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security'18)*.

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

[48] Jeong Wook Oh. 2010. Exploit Spotting: Locating Vulnerabilities Out of Vendor Patches Automatically. Black Hat USA.

[49] Jeong Wook Oh. 2011. DarunGrim: A Patch Analysis and Binary Diffing Tool. http://www.darungrim.org/.

[50] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 2019. 1dVul: Discovering 1-day Vulnerabilities through Binary Patches. In *Proceedings of the 49th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'19)*.

[51] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 226–237.

[52] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 504–515.

[53] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2009. DARWIN: An Approach for Debugging Evolving Programs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM.

[54] Ashwin Ramaswamy, Sergey Bratus, Sean W. Smith, and Michael E. Locasto. 2010. Katana: A Hot Patching Framework for ELF Executables. In *Proceedings of the 2010 International Conference on Availability, Reliability and Security*.

[55] Stephen Sims. 2016. Bruh! Do you even diff?—Diffing Microsoft Patches to Find Vulnerabilities. RSA Conference.

[56] Asia Slowinska and Herbert Bos. 2009. Pointless tainting? Evaluating the practicality of pointer tainting. In *EuroSys*. 61–74.

[57] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*.

[58] Temple F. Smith and M.S. Waterman. 1981. Identification of Common Molecular Subsequences. *Journal of Molecular Biology* 147, 1 (1981).

[59] Alexander Sotirov. 2006. Hotpatching and the Rise of Third-Party Patches. BlackHat USA.

[60] Zhenzhou Tian, Qinghua Zheng, Ting Liu, Ming Fan, Eryue Zhuang, and Zijiang Yang. 2015. Software Plagiarism Detection with Birthmarks Based on Dynamic Key Instruction Sequences. *IEEE Transactions on Software Engineering* 41, 12 (2015).

[61] Windows Unicorn vulnerability exploited in the wild. [online]. Windows Unicorn vulnerability exploited in the wild. https://securityaffairs.co/wordpress/30402/security/unicorn-exploited-in-the-wild.html.

[62] Harsimran Walia. 2011. Reversing Microsoft patches to reveal vulnerable code. Nullcon.

[63] Shuai Wang and Dinghao Wu. 2017. In-memory Fuzzing for Binary Code Similarity Analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*.

[64] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Behavior Based Software Theft Detection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*.

[65] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2019. Detecting" 0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 485–492.

[66] Ye Wang, Na Meng, and Hao Zhong. 2018. An Empirical Study of Multi-Entity Changes in Real Bug Fixes. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*.

[67] Dasarath Weeratunge, Xiangyu Zhang, William N Sumner, and Suresh Jagannathan. 2010. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 253–264.

[68] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20)*.

[69] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[70] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*.

[71] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*.

[72] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: Security Patch Analysis for Binaries Towards Understanding the Pain and Pills. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*.

[73] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploit. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS'17)*.

[74] Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 887–902.

[75] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. 2018. Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*.

[76] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS'19)*.

APPENDIX

A OUTPUT RESULTS FROM BINDIFF

For the example in Figure 1, we leverage BinDiff to compare the dynamic execution traces of the two ghttpd 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.

	ghttpd-1.4.3	ghttpd-1.4.4
1	xor %eax,%eax	xor %eax,%eax
2	lea 0x374(%esp),%eax	
3	mov %eax,0x10(%esp)	mov %bx,0x3b(%esp)
4	mov 0x370(%esp),%eax	mov $0x2e64,0x27(%esp)
5	lea 0xbd(%esp),%esi	
6	mov %bx,0x3b(%esp)	mov %edi,(%esp)
7		mov $0x252e,0x2b(%esp)
8		mov $0x5b20,0x2f(%esp)
9		mov $0x253a,0x33(%esp)
10		mov $0x5e25,0x37(%esp)
11		call strlen
12	lea 0x20(%esp),%ebx	lea 0x400(%eax),%ebx
13	mov $0xc8,0x8(%esp)	mov %ebx,(%esp)
14		call malloc
15	mov $0x1,0x4(%esp)	mov %ebp,0x14(%esp)
16		lea 0x3d(%esp),%ebp
17	mov %eax,0xc(%esp)	mov %edi,0x10(%esp)
18	mov %esi,(%esp)	mov %ebp,%edi
19	mov $0x2e64,0x27(%esp)	mov %ebx,0x4(%esp)

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.

804abbb	mov	%ebx,(%esp)	W@0xbfffb360[0x0804b34c]
804abd9	call	strlen	W@0xbfffb35c[0x0804abde]
804abe3	mov	%eax,(%esp)	W@0xbfffb360[0x00000426]
804abe6	call	malloc	W @0xbfffb35c[0x0804abeb]
804abeb	mov	%ebx,(%esp)	W@0xbfffb360[0x0804b34c]
804abee	mov	%eax,-0x1b0(%ebp)	W@0xbfffb388[0x08051da8]
804abf4	call	strlen	W@0xbfffb35c[0x0804abf9]
804abf9	mov	-0x1b0(%ebp),%edi	R@0xbfffb388[0x08051da8]
804ac15	mov	%edi,(%esp)	W@0xbfffb360[0x08051da8]
804ac25	mov	%eax,0x4(%esp)	W@0xbfffb364[0x00000426]
804ac29	call	vsnprintf	W@0xbfffb35c[0x0804ac2e]
804ac75	call	strftime	W@0xbfffb35c[0x0804ac7a]
804ac7a	mov	-0x1b0(%ebp),%edi	R@0xbfffb388[0x08051da8]
804ac80	mov	%edi,(%esp)	W@0xbfffb360[0x08051da8]
804ac83	call	strlen	W@0xbfffb35c[0x0804ac88]
804ac90	mov	%eax,(%esp)	W@0xbfffb360[0xbfffb492]
804ac93	call	strlen	W@0xbfffb35c[0x0804ac98]
804ac9c	mov	%eax,(%esp)	W@0xbfffb360[0x0000011c]
804aca5	call	malloc	W@0xbfffb35c[0x0804acaa]
804acaa	mov	%edi,0x14(%esp)	W@0xbfffb374[0x08051da8]
804acd2	mov	%eax,(%esp)	W@0xbfffb360[0x08052230]
804acd5	call	sprintf	W@0xbfffb35c[0x0804acda]
804ad24	mov	-0x1b0(%ebp),%eax	R@0xbfffb388[0x08051da8]
804ad2a	mov	%eax,(%esp)	W@0xbfffb360[0x08051da8]
804ad2d	call	free	W@0xbfffb35c[0x0804ad32]
804ad32	mov	%esi,(%esp)	W@0xbfffb360[0x08052230]
804ad35	call	free	W@0xbfffb35c[0x0804ad3a]
804ad54	pop	%ebx	R@0xbfffb52c[0xbfffb577]
804ad55	pop	%esi	R@0xbfffb530[0x00001000]
804ad56	pop	%edi	R@0xbfffb534[0x00000003]
804ad57	pop	%ebp	R@0xbfffb538[0xbffff6c8]
804ad58	ret		R@0xbfffb53c[0x0804a0f7]

Figure 6: A partial sequence of differences returned by BLEX. Each item includes the instruction, its operands, and the corresponding memory access during the instruction execution. Take the item in the first line, *W@0xbfffb360[0x0804b34c]*, for illustration. *W* refers to writing to memory. *0xbfffb360* refers to the memory address, and *0x0804b34c* is the value written to the memory cell at *0xbfffb360*.

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.

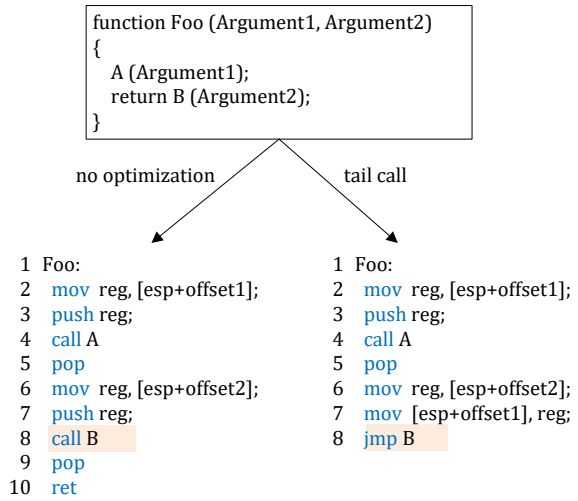


Figure 7: Tail call optimization example.

D SMITH-WATERMAN ALGORITHM

The input to Smith-Waterman algorithm is two sequences $A = a_1, a_2, ...a_n$ and $B = b_1, b_2, ...b_m$ of length n and m respectively. A maximum similarity matrix H is filled using the equation below.
$H_{(i,0)} = 0, 0 \leq i \leq m$,
$H_{(0,j)} = 0, 0 \leq j \leq n$,
and

$$H_{(i,j)} = Max \begin{Bmatrix} 0 \\ H_{(i-1,j-1)} + Sim(a_i, b_j) \\ Max_{k \geq 1} H_{(i,j-k)} + W_k \\ Max_{l \geq 1} H_{(i-l,j)} + W_l \end{Bmatrix} \quad (1)$$

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

Here A and B are two memory object access sequences. $Sim(a_i, b_j)$ 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 `cmp`. 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. Then, 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 overflowing this particular input field can trigger the vulnerability, and we confirm this in our PoC exploit.

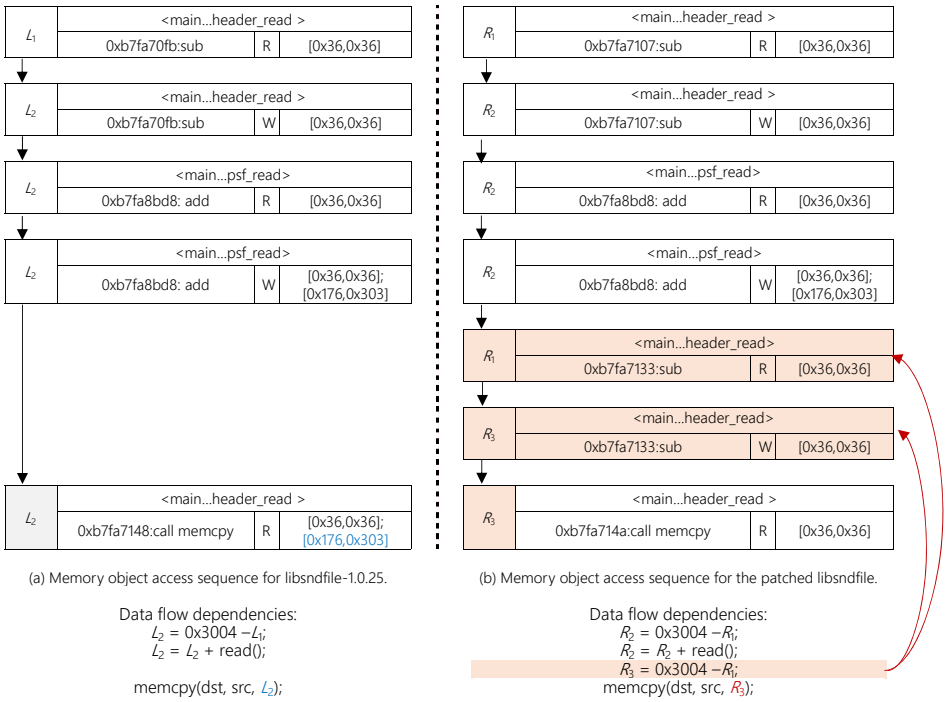


Figure 8: PatchScope's comparison result for libsndfile.