Capturing Invalid Input Manipulations for Memory Corruption Diagnosis

Lei Zhao, Keyang Jiang, Yuncong Zhu, Lina Wang, and Jiang Ming

Abstract—Memory corruption diagnosis, especially at the binary level where all high-level program abstractions are missing, is a tedious and time-consuming task. Given a crash, memory corruption diagnosis is expected to not only locate the root cause of the vulnerability, but also deliver rich semantics to understand the vulnerability. However, existing techniques can barely satisfy the above requirements. In this article, we present MemRay, a dynamic memory corruption diagnosis technique. The insight behind our approach is that most memory corruption is caused by malformed inputs, which further leads the vulnerable program to manipulate inputs by referencing invalid data structures. We design the "data structure reference sequence" to characterize how a program references various data structures to manipulate program inputs. Then, we identify memory corruptions by detecting violations in the input manipulations via data structures. We demonstrate the effectiveness of MemRay on a wide range of memory-corruption vulnerabilities. The result shows that MemRay precisely locates the root cause of vulnerabilities. Moreover, the "data structure reference" enables MemRay to deliver rich semantics and context information to assist vulnerability diagnosis on binary code.

Index Terms—Software security, crash analysis, memory corruption diagnosis

1 Introduction

Memory is the field of eternal arms races between attacks and defenses [43], [44]. Among types of software bugs, memory-corruption vulnerabilities have always been the research focus, and attacks by exploiting such vulnerabilities are always considered one of the top threats to cyberspace. [4], [43], [44].

Given an execution crash caused by invalid memory access (i.e., segment fault), memory-corruption diagnosis aims to locate the root cause of the underlying memory-corruption vulnerability, and figure out informative details about the context for reasoning the crash [38], [51], which is a common practice and complements multiple techniques in both defense and offense applications. For example, researchers have proposed a plethora of fuzzing techniques [20], [33], [42], [56] to detect memory corruption vulnerabilities in both academic and industry. However, fuzzing can only detect failed behaviors such as crashes. Analysts still rely on memory corruption diagnosis to locate the root cause, identify trigger conditions, and finally fix vulnerabilities. Besides, security experts should often pay close attention to new vulnerabilities, so that they can assess the security severity [25], [47], generate hot patches [36], [41] and honey patches [36], [41], and construct new detection rules [9], [29]. All of these applications rely on diagnosing a proof-of-concept of the newly disclosed vulnerability [18].

Despite the significance in multifaceted security applications, memory-corruption diagnosis is challenging, which is an error-prone process [35], [54] and may take a significant amount of time and domain knowledge [3]. To automate this process, both software engineering and security communities have proposed many techniques [3], [7], [34], [45], [48], [49], [50]. However, we argue that existing techniques do not suffice for memory-corruption diagnosis. In particular, we identify three distinct requirements: accuracy in locating the root cause, higher-level program abstractions to interpret memory corruption, and providing rich semantics for understanding vulnerability.

First, memory-corruption diagnosis should accurately detect the memory corruption and locate the root cause. In general, the root cause of memory corruption starts from an invalid pointer [43]. However, most of existing techniques identify memory corruption by detecting symptoms of memory corruption. For example, as memory corruption often leads to invalid data dependencies, CREDAL [49] identifies invalid data dependencies by tracking the data flow. Ravel [7] first constructs def-use pairs of pointers between instructions from passed executions, and then identifies invalid def-use pairs in a failed execution as suspicious vulnerabilities. The main limitation of such types of techniques is that it is difficult to distinguish which pointer in a def-use pair is the root cause.

Second, memory-corruption diagnosis should provide higher-level program abstractions to interpret the occurrence of memory corruption. At present, an increasing number of third-party organizations or individuals participate in the "the power of the crowd" for vulnerability identification [30]. Unlike software vendors who can diagnose a vulnerability with the support of source code, the third-party security analysts only have access to the binary code of commodity software. Therefore, memory-corruption diagnosis

• *Lei Zhao, Keyang Jiang, Yuncong Zhu, and Lina Wang are with the School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China. E-mail: {leizhao, kyjiang, yuncongzhu, lnwang}@whu.edu.cn.*
• *Jiang Ming is with Computer Science and Engineering Department, UT Arlington, Arlington, TX 76019 USA. E-mail: jiang.ming@uta.edu.*

Manuscript received 15 December 2020; revised 24 October 2021; accepted 17 January 2022. Date of publication 25 January 2022; date of current version 14 March 2023.
This work was supported in part by the National Natural Science Foundation of China under Grants 62172305, U1836112, and 61872273.
(Corresponding author: Lei Zhao.)
Digital Object Identifier no. 10.1109/TDSC.2022.3145022

often directly works on binary code, where high-level program abstractions are missing. For example, after we have identified an invalid pointer, a more existing question is which data structure this invalid pointer belongs to or what its size is. Unfortunately, existing binary-level techniques can only identify the root cause of memory corruption as low-level instructions. For instance, POMP [50] reverse executes the program from a crash point with forward-and-backward analysis, then reconstructs the data flow and pinpoints the critical instructions leading up to the crash. However, such types of techniques can't provide more high-level program abstractions or explanations of the vulnerability.

Third, as the ultimate goal of memory-corruption diagnosis is to fix the vulnerability, memory-corruption diagnosis should provide rich semantics to help understand how memory corruption is triggered. To trigger memory corruption, an execution context has to satisfy specific conditions, which are related to both program semantics and program inputs. Rich semantic information, such as unsafe input fields, vulnerable data structures, and memory operations that lead to corruption, is required to understand the memory corruption. This process has been proven to be very complicated at the source code level [49], let alone at the binary level. For example, AURORA [3] generates a lot of predicates and then leverages coverage statistics to calculate and prioritize suspicious predicates contributing to crashes. A key challenge in statistical debugging techniques is that they may locate predicates corresponding to both root causes and infected states [55]. Moreover, these predicates are inefficient in vulnerability diagnosis due to the lack of high-level semantic information.

Besides, memory-corruption diagnosis with rich semantics can also benefit automatic security patch generation [26] and vulnerability repair [21], [23], [39]. A patch for fixing the vulnerability consequently depends on specific program logic because vulnerability is mostly caused by a specific programming logical fault. Take a classic memory operation, such as *memcpy (dst, src, size)*, for illustration. Memory corruption could be caused by a fault in all three parameters. Memory-corruption diagnosis with rich semantics could facilitate the process of fix localization [39].

Our Approach. In this paper, we propose MemRay, a dynamic memory-corruption diagnosis technique by capturing invalid input manipulations. Our approach is motivated by the following two observations.

O1: Program inputs (e.g., network protocols, images, and digital media files) usually contain rich structural information. When receiving an input, a program subsequently allocates various data structures such as variables, arrays, and `structs`, to manipulate the multiple input fields [5]. The way that a program references various data structures to manipulate program inputs reveals a wealth of program semantics.

O2: As most memory-corruption vulnerabilities are triggered by unsafe inputs, execution on such unsafe inputs will violate the program semantics by referencing an invalid data structure to manipulate program inputs. Therefore, capturing the violations between the manipulations on input fields and corresponding data structures brings a promising approach to identify memory corruption.

Based on these two observations, we design a novel semantics-aware representation, *data structure reference sequence*, to represent how a program references various data structures to manipulate input fields during dynamic execution. Each element, denoted as a *data structure reference*, refers to manipulation on inputs via the corresponding data structure. To locate the root cause, we identify memory corruption by detecting the following violations: the manipulation on program inputs exceeds the boundary of the corresponding data structure, or it is performed by referencing an invalid data structure. To address the challenge that high-level program abstractions are missing in binaries, we dynamically excavate data structures as well as program input formats at run time. In detail, we leverage a fine-grained dynamic taint analysis to monitor execution and track the propagation of program inputs, excavate both data structures and input fields, construct the *data structure reference sequence*, and finally identify memory corruption by detecting invalid *data structure reference*. Further, invalid *data structure reference* can deliver semantic information about both vulnerable data structures and input fields.

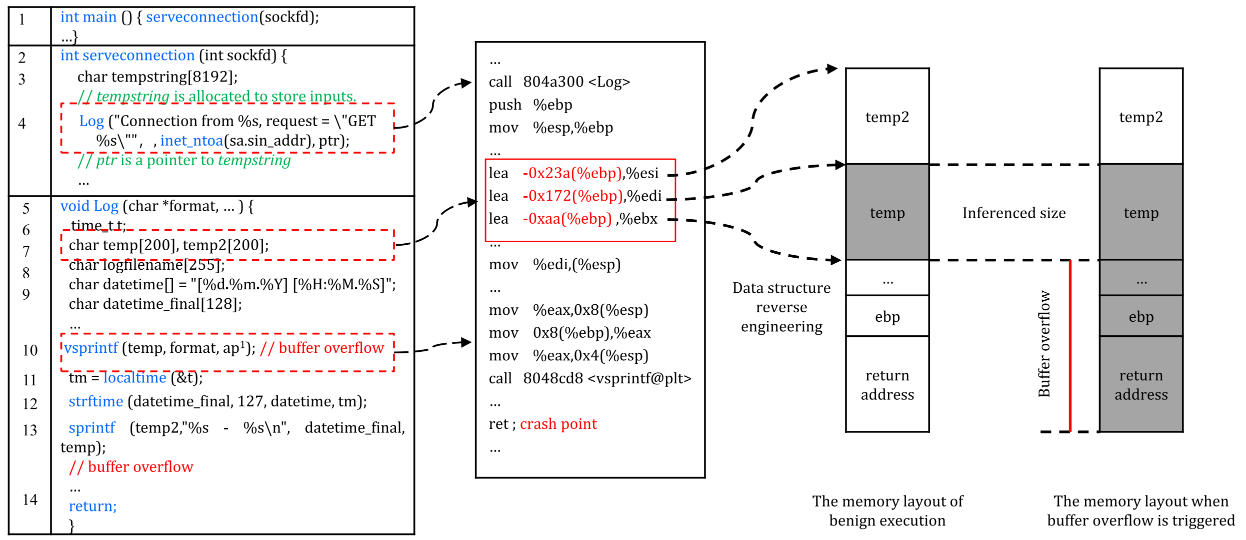
We demonstrate the effectiveness of MemRay on a wide range of memory-corruption vulnerabilities. Our evaluation shows that MemRay can precisely locate vulnerability root cause from tediously long traces. In addition, rich semantics provided by MemRay can identify contexts to trigger the vulnerability and specific input fields that are correlated with that vulnerability, which greatly reduces the amount of effort required for diagnosing vulnerabilities.

Contributions. In summary, we make the following contributions:

- *Novel representation for program semantics*. We propose *data structure reference sequence*, a semantics-aware program representation to characterize how a program references data structures to manipulate program inputs. This representation is particularly fit for vulnerability diagnosis.
- *Accuracy and interpretability in memory-corruption diagnosis*. Taking advantage of reverse engineering of program data structures and input formats, we propose to locate vulnerability root cause by capturing the violations between input manipulation and corresponding data structure. Our approach reveals rich semantics (e.g., vulnerable data structures and unsafe input fields that cause memory corruption) to locate the root cause and understand vulnerabilities.
- *Open-source implementation*. We build a prototype that can directly work on stripped binaries. The source code of MemRay is available at https://github.com/Ma5ker/MemRay.git. Evaluations on real-world vulnerabilities show that our approach is effective in diagnosing vulnerabilities. Moreover, MemRay diagnoses and discovers an unknown vulnerability, which is confirmed by CVE (CVE-2021-25794 [13])

2 Problem Statement and Our Insight

In this section, we present the problem statement of our research and share our insights.



```
1   int main () { serveconnection(sockfd);
    ...}
2   int serveconnection (int sockfd) {
3     char tempstring[8192];
      // tempstring is allocated to store 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, ... ) {
6     time_t t;
7     char temp[200], temp2[200];
8     char logfilename[255];
9     char datetime[] = "[%d.%m.%Y] [%H:%M.%S]";
      char datetime_final[128];
      ...
10    vsprintf (temp, format, ap¹); // buffer overflow
11    tm = localtime (&t);
12    strftime (datetime_final, 127, datetime, tm);
13    sprintf  (temp2,"%s  -  %s\n",  datetime_final,
    temp);
      // buffer overflow
      ...
14    return;
    }
```

```
...
call  804a300 <Log>
push  %ebp
mov   %esp,%ebp
...
lea   -0x23a(%ebp),%esi
lea   -0x172(%ebp),%edi
lea   -0xaa(%ebp) ,%ebx
...
mov   %edi,(%esp)
...
mov   %eax,0x8(%esp)
mov   0x8(%ebp),%eax
mov   %eax,0x4(%esp)
call  8048cd8 <vsprintf@plt>
...
ret ; crash point
...
```

temp2
temp
...
ebp
return address
Data structure reverse engineering
Inferenced size
Buffer overflow
temp2
temp
...
ebp
return address
The memory layout of benign execution
The memory layout when buffer overflow is triggered

¹ *ap* in Line 10 is the type of *va_list*, which stores a variable arguments list from program inputs.

(a) A code snippet of ghttpd-1.4.3 (b) Corresponding instructions (c) Memory Layout

Fig. 1. Motivating example.

2.1 Problem Statement

For a formal problem statement, we use P to denote a program that contains a memory-corruption vulnerability; I denotes an unsafe input on which the execution of P results in a crash. Memory-corruption diagnosis aims to locate the root cause and provide rich semantics for understanding the vulnerability. We design MemRay to achieve the following goals:

- *No reliance on source code. P* is a stripped binary program, of which the source code is not available or the debugging information has been stripped off.
- *Accuracy in locating the root cause.* MemRay can accurately locate the root cause of the underlying vulnerability in a crash, e.g., instructions that result in memory corruption.
- *Interpretability.* MemRay can provide higher-level program abstractions with rich semantics to answer the questions of how to trigger the vulnerability and how to fix it.
- *High robustness.* MemRay is expected to handle both *spatial errors* (e.g., buffer overflow and format string) and *temporal errors* (e.g., use-after-free and dangling pointer dereference).

Our study assumes that I is available. I could be an input generated by fuzzing or a PoC from vulnerability reports. With such an assumption, the problem scope of our study is different from that of crash analysis techniques that work on core dumps [10], [11], [31], [48], [49], [50].

Core dumps are automatically generated by an operating system to capture the snapshot of the memory when a process has crashed. The advantage of core dump analysis is that it is lightweight for the diagnosis of program crashes, as it does not require program instrumentation, nor rely upon the log of program execution.

However, core dump analysis is insufficient in diagnosing a vulnerability due to the partial program states carried by core dumps. That is, a core dump only provides a memory snapshot of an execution failure, from which core dump analysis techniques can only infer partial control and data flows to program crashes. For example, the design goal of POMP [50] is to highlight and present to software developers (or security analysts) the minimum set of instructions that contribute to a program crash, instead of the precise root cause instructions. By contrast, vulnerability diagnosis in our study aims to locate the root cause and provide higher-level program abstractions with rich semantics to understand the vulnerability.

2.2 Motivating Example

We use a real-world vulnerability CVE-2001-0820 [12] in `ghttpd-1.4.3` to motivate our approach as well as to interpret technical details in the following sections. Please note that we present the source code to ease representation, and our approach works on binary code directly.

As shown in Fig. 1, the program first receives a URL, stores it into the buffer `tempstring`, and defines a pointer *ptr* to dereference `tempstring`. The function `Log` transfers the URL into a formatted string, and saves it into `temp` and `temp2` of length 200-bytes. There is a stack overflow introduced by calling `vsprintf`, which will overwrite the data on the stack. If the program receives an input with a long string (larger than 200-bytes), a memory corruption will occur.

This vulnerability is a little tricky to diagnose. First, the crash point is at address `0x804a41c`, which is far away (up to 23,395 instructions) from the root cause. Among such long execution traces, there are multiple instructions that perform memory operations. Thus, it is difficult to identify which operation results in memory corruption. Second, there are two corruption points. Without the support of high-level program abstractions, an analyst can identify the crash in `call vsprintf` which tampers with the return address. However, he may miss the section corruption point in `call sprintf`. Actually, the official patch for this vulnerability is incomplete, which only fixes one of the two corruption points. MemRay can detect this spatial memory error easily by reversing the data structures and identifying the input manipulation via the data structures, as shown in Figs. 1b and 1c. We will present more details in the following sections.

2.3 Our Insight and Challenges

Based on the aforementioned two observations, detecting violations between input manipulations among data structures enables us to identify invalid pointers and locate the root cause. For the example in Fig. 1, multiple variables are used to manipulate program inputs (*tempstring*, *temp*, *temp2*). To manipulate program inputs, the program should dereference a pointer of which the address is calculated by adding an offset to the base address of a data structure. If the program receives an input of which the length exceeds the size of data structures (*temp* and *temp2*), the manipulation on the input will result in an invalid pointer with the increasing offset.

To implement our observation and insight, there are several technical challenges. First, all high-level program abstractions are missing in binaries. A second challenge is how to construct the corresponding relationship between the data structure reference and input manipulation. Third, the detailed principles to identify memory corruption is still unknown. As there are types of memory corruption vulnerabilities, including spatial and temporal memory corruption, the root cause may vary a lot for types of vulnerabilities.

3 Definition and Overview

3.1 Definition

We dynamically reverse engineer data structures together with their context information. Therefore, we split our definition into a reverse-engineered data structure *var* and the runtime reference to *var*.

Definition 1. Reverse-engineered Data Structure. *A reverse-engineered* `data structure` var = (alloc, size, type, v). alloc *refers to the context information when allocating the data structure*, size *indicates the size (in byte) of the data structure*, type *means its type, and* v *indicates the alive status.*

In our study, *type* includes static variables, dynamically allocated variables in the heap, and local variables in the stack. As allocation context varies from different types to data structures, the definition of *alloc* has to be considered together with the program context where *var* is allocated.

- For a static variable, as its memory slot will not be reused, its memory address can exclusively represent *alloc*, and the alive status of v is always true.
- 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 that compiler optimization options may allocate registers to hold local variables for better performance, especially for variables with the type `integer`. As there is no base address for register variables, we use the related registers to represent such specific local variables. For a local variable, v is true if its function frame exists. Otherwise, v is false.
- 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*. The v for a dynamic variable is true if the allocation is valid and alive, and it becomes false when the variable is freed.

Take the running example from Fig. 1 to illustrate Definition 1. The function `Log` defines a local variable `temp` with 200-bytes length. The `alloc` for `temp` will include the calling context, `main-serveconnection-Log`, as well as the offset from the frame pointer of `Log`.

Definition 2. Data Structure Reference. *The operation of referencing a data structure to manipulate input fields at run time is denoted as* A(var) = (var, cc, op, optype, α, offset), *where:*

- var *is the reverse-engineered data structure as defined by Definition 1.*
- cc *refers to the context in which* var *is referenced.* cc *includes the function that references* var *as well as the calling context of this function. Please note that* cc *for* var *may not be identical with the context where* var *is allocated, because a variable can be referenced by other functions through pointers. Following the def-use principle, we denote* alloc *as defined by Definition 1 to indicate the context in which* var *is allocated, and define* cc *to represent the context where* var *is used.*
- op *refers to the data-flow-related operations on* var. *We mainly focus on two main types of instructions that are involved in the explicit input propagation: data movement instructions and calling instructions to library/system calls. We denote* op *as the instructions as well as their addresses.*
- optype *contains two reference types of* var*: read and write.*
- α *refers to input fields that are manipulated via* var.
- offset *refers to the interval of the specific region in* var *that is used to manipulate* α.

For the invocation of memory-allocation functions, we particularly generate the data structure reference. Temporal memory errors such as use-after-free and double-free can also result in a crash without manipulating program inputs in memory-allocation functions. For example, it is common that a dangling pointer to a freed object points to raw data, and taking this dangling pointer as parameters to invoke `free` may result in a crash. If the invocation of memory-allocation functions does not manipulate program inputs, then only *var* and *cc* in the data structure reference make sense.

Please note that α may not stay the same as the program subsequently parses the input, especially for the input with nested structures. As shown in Fig. 2, when `ghttpd` receives an input with 30-byte length, it allocates a large buffer `tempstring` to store the input. Thus the correlated input with `tempstring` is the whole input, of which the

```
recv(sockfd, tempstring, 4096, 0);
// the correlated input of tempstring ranges [0, 29]
ptr = strtok(tempstring, " ");
if(strcmp(ptr, "GET"))
// the correlated input of ptr ranges from [0, 3]
ptr = strtok(NULL, " ");
Log("GET %s\", ptr);
// the correlated input of ptr ranges from [4, 29]
```

Fig. 2. The procedure of parsing ghttpd's inputs.

offset ranges from 0 to 29. Then, as `ghttpd` defines another pointer `ptr` to dereference a part of the input, the correlated input offset with `ptr` ranges from 0 to 3. At last, the correlated input offset with `ptr` becomes 4 to 29 when calling `Log`.

3.2 Rules to Identify Memory Corruption

According to Definitions 1 and 2, we identify memory corruption with the following two principles.

R1: During the dynamic execution, we track the liveness of a data structure, as indicated by v in (*alloc*, *size*, *type*, v). For a data structure reference (*var*, *cc*, *op*, *optype*, α, *offset*), we identify memory corruption if *var* is not alive (e.g., *var* is freed or the function frame holding *var* does not exist).

R2: For a data structure reference (*var*, *cc*, *op*, *optype*, α, *offset*), *offset* refers to the interval of the specific region in *var* that is used to manipulate α. With *offset*, we can further inspect that the specific region does not exceed the boundary of *var*, as indicated by *size* in (*alloc*, *size*, *type*, v). Otherwise, we identify memory corruption of overflow.

We leverage the principle **R1** to detect memory corruption caused by temporal memory errors (such as use-after-free and double-free), and leverage **R2** to detect memory corruption caused by spatial memory errors (such as buffer overflow). For vulnerability diagnosis, MemRay delivers rich semantics via items in the data structure reference (*var*, *cc*, *op*, *optype*, α, *offset*), such as the vulnerable data structure, its boundary, and input fields leading to memory corruption.

4 Design and Implementation

4.1 Overview

The overview of MemRay is shown in Fig. 3. It mainly contains four steps: dynamic tainting and execution monitoring, data structure excavation, data structure reference construction, and vulnerability analysis.

First, we leverage a fine-grained dynamic tainting technique to monitor dynamic executions of P on I. The fine-grained dynamic tainting gives each input byte a unique taint tag. The execution monitoring records the propagation of all tainted bytes, as well as the executed instructions and their operands.

Second, inspired by the previous work [40], we reverse engineer program data structures from execution traces. The novelty of our technique is we reverse engineer both data structures and input fields correlatively. This approach enables us to construct the correlation between data structures and input fields.

Third, we construct data structure reference sequences to represent how a program references various data structures to manipulate an input during execution.

At last, we identify memory corruption by detecting invalid data structure references during the manipulation of program inputs. With rich semantics in the representation of data structure reference, our approach can further identify the vulnerable data structure, corresponding input field that results in memory corruption for understanding the vulnerability.

4.2 Dynamic Tainting and Execution Monitoring

We build the dynamic taint analysis and execution monitoring based on DECAF [24], a whole-system dynamic binary analysis platform. With the fine-grained dynamic tainting, we give each input byte a unique taint tag and then track the propagation of every input byte. 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 taint tags of all source operands.

We monitor all executed instructions in both the targeted program and invoked library functions. The reason is that many memory corruption is caused by the misuse of library functions (such as `vsprintf`), and such fine-grained monitoring enables us to capture mistake manipulations. Besides, we record runtime information used in the data structure extraction. For example, we hook related memory allocation system/library calls (e.g., `mmap`, `malloc`, and `malloc`) to track dynamically allocated variables on the heap.

Fig. 4 shows a fragment execution trace on the `ghttpd-1.4.3` in Fig. 1. The execution trace presents details of dynamic execution information, including instructions, operands, and their values, manipulated inputs with taint tags. In the following sub-sections, we will use this fragment trace as an example to present our implementation details.

4.3 Excavating Program Data Structures

Reverse engineering program data structures and input fields in this study is inspired by the previous work, Howard [40]. We share a similar insight with Howard in

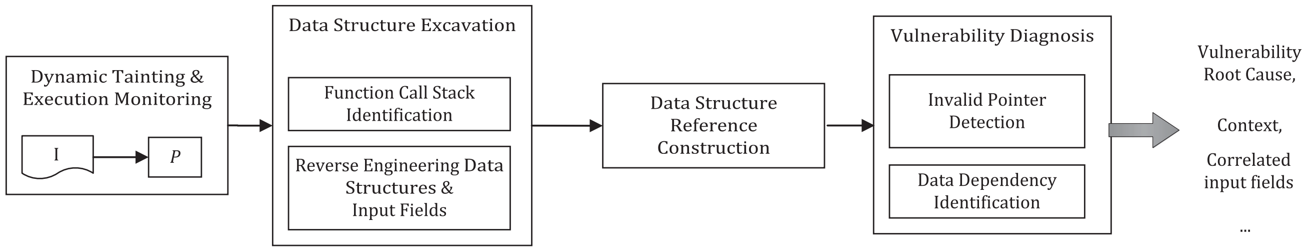

Dynamic Tainting & Execution Monitoring
I
P
Data Structure Excavation
Function Call Stack Identification
Reverse Engineering Data Structures & Input Fields
Data Structure Reference Construction
Vulnerability Diagnosis
Invalid Pointer Detection
Data Dependency Identification
Vulnerability Root Cause,
Context,
Correlated input fields
...

Fig. 3. MemRay system overview.

```
804a314    lea        -0x172(%ebp), %edi          A@0xbfffb426[0x0000000](R)          T0
804a31a    mov        %edi, (%esp)                M@0xbfffb240[0xbfffb426](W)         T0
804a351    mov        %eax, 0x8(%esp)             M@0xbfffb248[0xbfffb5a4](W)         T0
804a35c    call       0x8048cd8 (vsprintf)        M@0xbfffb23c[0x804a361](W)          T0
……
b7ef952c   rep movsl  %ds:(%esi),%es:(%edi)       M@0xbfffb5dd[0x41414141](R)         T1{6; 7; 8; 9;}
                                                  M@0xbfffb456[0x41414141](W)         T1{6; 7; 8; 9;}
```

For M@0xbfffb5dd[0x41414141](R) T1{6; 7; 8; 9;}, "M" indicates a memory operand; "0xbfffb5dd[0x41414141]" indicates that the memory address of the memory cell and its value; "R" indicates READ; "T1" indicates this operand is tainted, and taint tags for the four bytes in the operand are followed, {6; 7; 8; 9;}, which are separated with semicolon.

Fig. 4. A fragment of dynamic execution traces.

that memory access patterns reflect the types of program data structures. Our difference with Howard is that we need to establish the correlations between reverse-engineered data structures and input fields. For this requirement, our approach utilizes taint analysis to identify only data structures that are used to manipulate program inputs, instead of all of the data structures that are referenced during execution.

4.3.1 Function Call Stack Identification

Recall that data structures are allocated and referenced dynamically along with execution. We leverage the calling context to represent when and where a data structure is allocated or referenced. As the calling context changes along with stack frames, our first step is to identify the function call stack. In this study, we identify stack frames by matching `call/ret` pairs and the balance of stack pointers when the paired `call/ret` is executed.

This straightforward method can deal with most of the cases, except for two compiler optimization options, tail call optimization [8] and function inlining. The tail call optimization [8] replaces the tail inter-procedural `call` with `jmp`. The function inlining leads to no explicit control flow transition. Our solution is to treat each of the counterexamples as an extended function frame. We argue that our treatment does not affect MemRay's performance, because the inaccuracies due to tail-call or function inlining optimization will not bring a negative impact on data structures.

4.3.2 Root Pointer Identification

Typically, the allocation of a data structure returns a pointer for reference. We denote this pointer as a *root pointer*. In general, root pointers are unique for different data structures, and they are not derived from any other root pointers. Based on this definition, we extract data structures by extracting their root pointers. For the three types of data structures (see Definition 1), we extract their root pointers in different ways.

Identifying the root pointer for a static variable is simple because its memory address will not be reused. Thus, we directly use the memory address to represent its root pointer.

For dynamic variables on the heap, we hook related memory allocation functions and take the return value as the root pointer. Typically, `eax` holds the return value of a memory allocation function, which refers to the root pointer of the dynamic variable.

Identifying the root pointer for a local variable is a little more tricky. A function stack frame generally 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 -0x10(%ebp), %ebx`) in Fig. 5 indicates a root pointer wit the address `ebp-0x10`, because it is in the current stack frame and is not derived from any other root pointers. With this approach, we extract six local variables as shown in Fig. 5.

A challenge is that a local variable can also be addressed via the stack pointer `esp`, which may lead to false negatives on data structure extraction. To address this concern, we normalize to reference local variables only via `ebp`. Specifically, the concrete execution state enables us to examine whether two pointers, one addressed via `esp` and another one addressed via `ebp`, are *aliased*.

4.3.3 Tracking Pointer Propagation

With root pointers, we can identify the allocation of data structures. After that, we extract the references to these data structures via tracking the propagation of root pointers. Given all of the root pointers identified in the last step, we first identify *alias* pointers by tracking the data movement of root pointers. Second, we track all arithmetic calculations and memory movements on root pointers. At last, for a memory load/store, we identify its root pointer by checking the data dependency of its base address. For example, the

```
804a314:  lea   -0x172(%ebp), %edi
804a31a:  mov   %edi,(%esp)
……
804a358:  mov   %eax,0x4(%esp)
804a35c:  call   8048cd8 <vsprintf@plt>

804a36c:  lea   -0xaa(%ebp), %ebx
804a377:  mov   %ebx,(%esp)
…
804a38c:  call   8048d38 <memset@plt>
804a391:  lea   -0x2a(%ebp), %eax
804a394:  mov   %esi,0xc(%esp)
804a398:  lea   -0x23a(%ebp), %esi
804a39e:  mov   %ebx,(%esp)
804a3a1:  mov   %eax,0x8(%esp)
804a3a5:  movl  $0x7f,0x4(%esp)
804a3ad:  call   8048fb8 <strftime@plt>
```

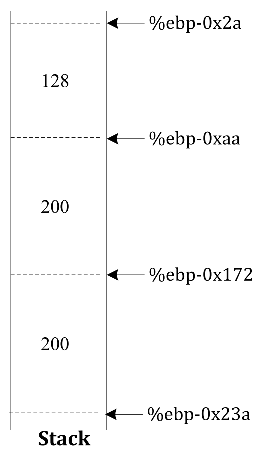


Fig. 5. Reverse-engineered data structures in ghttpd. The root pointers are colored as red.

address in an x86/x64 instruction is computed as *address = base + (index × scale) + offset*, where *base* means the base address. By tracking the propagation of root pointers, we can identify the pointer on which *base* depends as the root pointer.

For the example in Fig. 4, the root pointer (`0xbfffb426`) indicated by `-0x172(%ebp)` is used as a parameter for `call vsprintf`. For the memory cell accessed by the instruction at `0xb7ef952c`, we identify that its address `0xbfffb456` depends on `0xbfffb426`. Then, we can identify that `call vsprintf` references the data structure indicated by `-0x172(%ebp)`.

4.3.4 Size Inference

For a dynamically allocated variable, the parameter explicitly indicates the size. However, the obstacle is to infer the size for local variables and static variables, because their types are invisible in binary code. To address this challenge, we calculate the offset interval between two contiguous root pointers.

As shown in Fig. 5, the sizes of these data structures are just the offset interval between adjacent root pointers. Note that the size could be over-approximated if the adjacent data structure is not referenced during execution.

4.3.5 Identifying Input Fields

According to Definition 2, an input field refers to a sequence of bytes that propagate from program inputs and are manipulated via the same data structure. We identify input fields via tainted operands and excavated data structures.

To be more specific, we first identify all memory load/store instructions with tainted operands. With the memory address of tainted operands, we further identify the corresponding data structure to this tainted operands. Then, the taint tag of a tainted operand indicates the corresponding input bytes that propagate to this memory cell. In our design, the taint tag for each input byte is assigned as the offset from the beginning of the input. Therefore, we can identify continuous bytes according to their taint tags. At last, an input field is identified for a sequence of bytes if and only if the taint tags of these bytes are continuous and these bytes are manipulated via the same data structure.

4.4 Constructing Data Structure Reference

This subsection explains how we determine each item in the data structure reference model: (*var*, *cc*, *op*, *optype*, α, *offset*).

For a memory load/store operation of which the operands are tainted, we first identify *var* and α for the tainted memory cell based on data structure excavation as demonstrated above. Then, the *optype* of *read* or *write* for the related memory operand is decided by whether it is loaded from or stored into memory.

Second, we regard *cc* as the current calling context, and identify *op* for types of instructions. If the instruction belongs to the program itself, we regard *op* as the instruction. If the instruction belongs to a library function, we will regard *op* as the invocation of the library function. For example, we regard the *op* for the memory operation at `0xb7ef952c` in Fig. 4 as `call vsprintf` at `0x804a35c` instead of `rep movsl`. The basic insight is the library function is a complete unit and we only focus on the results after the invocation of the library function.

Third, we calculate the offset from the root pointer to the tainted memory cell, and use this offset to index the specific memory region. In this way, we denote the memory address offset to represent *offset* in Definition 2. In particular, it is common that a program may reference one data structure to manipulate multiple fields (e.g., a format string including multiple input fields). To distinguish them, we identify specific memory regions for different input fields. Specifically, we construct multiple data structure references to represent the manipulation of every input field. These input fields are manipulated via the same data structure, but they are stored in different memory regions of the data structure. Now, we can establish the correlations between data structures and input fields.

For input fields that contain multiple bytes such as a string, a program generally deals with them in a loop. As a consequence, an instruction may be executed multiple times. Actually, these repeated instructions deal with different bytes in the same fields. Therefore, we further group an execution sub-sequence for the operands deriving from the same input field. With this approach, we can further shrink the number of data structure references to facilitate analysis.

An exception is the *I/O related functions*, for which the memory operation by *syscalls* is implicit or cannot be captured with our dynamic tainting. To address this challenge, we retrieve such memory operation according to the parameters of such *I/O related functions*. In particular, we summarize semantics for *I/O related functions*. Then, we monitor the parameter values for calling these functions at run time. Finally, we retrieve the memory operation according to the function semantics and their parameters. For example, the first parameter for calling `recv` is a pointer to a data structure. The return value of `recv` indicates the number of bytes stored into memory. With such knowledge, we can retrieve the memory operation by `recv`.

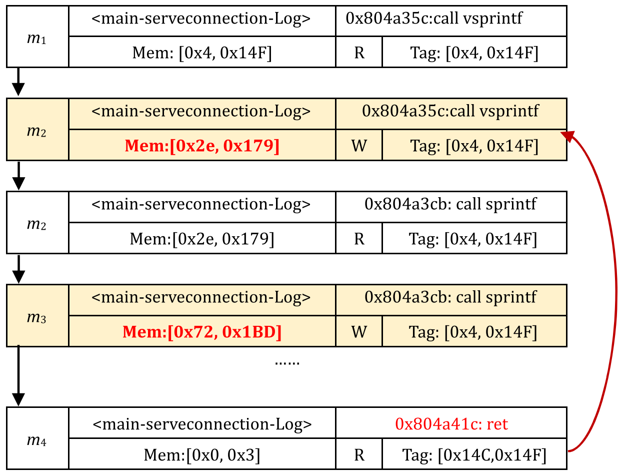
We take the example in Fig. 4 to summarize the work-flow to construct a data structure reference. First, we identify a root pointer, which is `ebp-0x172`. Then, we track the propagation of the root pointer to identify every memory load/store. Take the instruction at `0xb7ef952c` for illustration. Its destination operand with address `0xbfffb456` is tainted and addressed via `edi`. To identify the belonging data structure, we examine root pointers on which the memory address `0xbfffb456` is dependent. Then, we identify the value of `edi` is propagated from the root pointer `ebp-0x172`. Moving further, we identify the *optype* as `W`. The value `0x41414141` propagates from the input fields with the offset interval $[6-9]$ (α). The specific memory region (*offset*) of the destination operand ranges from `0x30`(*0xbfffb456-0xbfffb426*) to `0x34`. As the instruction at `0xb7ef952c` belongs to a library function, we regard the invocation of `call vsprintf` as *op*. Finally, we construct a data structure reference combined with the current calling context (*cc*).

4.5 Memory Corruption Diagnosis

As referencing a corrupted data structure is just the effect instead of the cause, we only identify the invalid pointer as memory corruption and ignore the following dereferences to this invalid pointer.

var	alloc	size	type	v
m_1	<main-serveconnection>:ebp-0x4151	0x2000	stack	1
m_2	<main...Log>: ebp-0x172	0xc8	stack	1
m_3	<main...Log>: ebp-0x23a	0xc8	stack	1
m_4	<main...Log>: ebp+0x4	0x4	stack	1

(a) Allocated memory objects in the crashed execution.



(b) Memory object access sequence in the crashed execution.

Fig. 6. Data structure reference sequences for the example in Fig. 1. Take the first item to interpret (var, cc, op, optype, α). "m1" (var) represents a reverse-engineered data structure in (a); "<main-serveconnection-Log>" (cc) is the calling context and function that references "m1"; "0x804a35c:call vsprintf" (op) refers to the instruction operating on "m1"; "Mem:[0x4, 0x14f]" (offset) refers to the memory region in "m1"that are used to store correlated input; "R" (optype) means the reference type of read; "Tag:[0x4,0x14f]" (α) indicates the input field that is manipulated via "m1". Items labeled as yellow indicate invalid memory references.

Compared with low-level instructions in binaries, semantic information in a *data structure reference* is more significant and useful for facilitating the vulnerability diagnosis. An invalid *data structure reference* consists of rich semantics including the vulnerable data structure, the input field that results in memory corruption, and corresponding memory regions used to manipulate the input field.

We further construct invalid data dependencies caused by memory corruption. This information is helpful to assess the severity or exploitability of the vulnerability. Following the memory read-write chain, a data dependency exists if the memory region that is used to write intersects with another one that is used to read.

Let us take the motivating example in Fig. 1 to illustrate the vulnerability diagnosis by putting all the above technical details together. Fig. 6a presents data structures allocated in the crash (Definition 1). Fig. 6b shows how the program references these data structures to manipulate inputs (Definition 2).

As shown in Fig. 6b, the invocation of `vsprintf` tampers with the return address of `Log`, and the mismatch data dependency is indicated by a red line. By examining the data structure reference in the invocation of `vsprintf`, we observe that the invocation of `vsprintf` manipulates the input field with the offset from $0x4$ to $0x14f$ via the data structure m_2. To be more specific, it stores the input field into the memory region in m_2 from offset $0x2e$ to $0x179$. As the root pointer of m_2 is $ebp - 0x172$, the memory addresses of the memory region with offset $[0x2e, 0x179]$ range from $ebp - 0x144$ to $ebp + 0x7$. Consequently, the return address (with address from $ebp + 0x4$ to $ebp + 0x7$) is overwritten.

Moving further, we identify a second invalid data structure reference in the invocation of `sprintf`. We can observe that it stores the input field into the memory region with offset $[0x72, 0x1bd]$. As the root pointer of m_3 is $ebp - 0x23a$, the memory addresses of the memory region with offset $[0x72, 0x1bd]$ range from $ebp - 0x1cb$ to $ebp - 0x7d$. We then observe that this memory region intersects with m_2, because the root pointer of m_2 is $ebp - 0x172$. Therefore, we identify the second memory corruption.

Actually, the second memory corruption cannot be exploited, because of the truncation by "00". However, the security patch `ghttpd-1.4.4` makes the second corruption exploitable. The patch fixes the first memory corruption point by changing the vulnerable local variables to dynamic variables in the heap. With this impact, the second memory corruption can overrun the data structure for heap memory management and finally results in a crash.

4.6 Execution Comparison to Reduce False Positives

Our approach may encounter false positives caused by specific program logic or compiler optimizations. For example, it is not rare that a program could assign multiple objects with the same value by only one operation (e.g., calling memset). Another common issue is compiler optimizations on complicated data structures (such as `struct`). Typically, the address of an internal item in a `struct` should be calculated by adding an offset to the root pointer of the `struct`. However, modern compilers often optimize the addressing of internal items with the frame pointer (*ebp*) directly. In this case, there is no distinguish between local variables and items in the `struct`, and our approach will excavate internal items in the `struct` as individual items. Then, the operation of referencing the entire `struct` will be detected as memory corruption.

To alleviate false positives, we leverage execution comparison to distinguish suspicious memory corruption caused by specific program logic or compiler optimizations. The basic insight is that such suspicious memory corruption also occurs in passed executions. Therefore, the execution comparison can counterbalance false positives.

Please note passed execution used for comparison does not necessarily follow the same execution trace as the crashed one. Previous techniques like [45] require a passed execution of which the execution trace is identical to that of the failed one until the vulnerability is triggered. However, it is not clear how often this second input exists, or how easy it is to create one, reducing the chance of applicability in practice. In our study, we only require the passed execution covering functions or statements involved in suspicious data structure references. Using coverage guided testing like fuzzing is practical to generate a passed input.

5 Evaluation

The prototype of MemRay consists of dynamic tainting for execution monitoring and an off-line trace analysis for vulnerability diagnosis. We extend DECAF [24] to support multi-tag taint analysis with an additional 560 lines of C code. Our off-line trace analysis includes 1,850 lines of Python code.

5.1 Experiment Setup

We conduct our experiments with two objectives. First, the evaluation on real-world vulnerabilities aims to show that MemRay can locate root cause from tedious traces. Second, the data structure reference sequence enables MemRay to deliver rich semantics for understanding vulnerabilities.

5.1.1 Evaluation on Known Vulnerabilities

We select vulnerabilities from a recent study [30], which contains 386 real-world vulnerabilities with various types, including format string vulnerability, use-after-free (UAF), data segment (BSS) overflow, and off-by-one error. Our selection follows the criteria: 1) the vulnerable program and a PoC are available; 2) the security patch is also available in relevant datasets [28], [30], [46]. With these criteria, 73 vulnerabilities are selected. Besides, we exclude 4 vulnerable applications (such as interpreted language engines) that cannot be supported by our prototype. For the remaining 69 vulnerabilities, we randomly select 35 (50%) for evaluation.

We select vulnerabilities for open-source programs with their patches available. In this way, we regard the security patches as the ground truth of vulnerability root cause. To be more specific, we manually examine the patch source code and identify the instructions of root cause.

5.1.2 Evaluation on Unknown Vulnerabilities

To demonstrate the effectiveness of MemRay in analyzing unknown vulnerabilities, we select multiple programs that are widely tested in recent fuzzing techniques [19], [52]. Then, we leverage AFL to test these programs for discovering crashes. Further, we leverage MemRay to diagnose these crashes and detect vulnerabilities.

5.1.3 Baseline Techniques

From the perspective of problem statements, we don't select crash analysis techniques that work on core dumps, such as POMP [50], POMP++ [31], REPT [10], and kernel REPT [22]), as baseline techniques. As demonstrated in Section 2, a core dump only provides a memory snapshot of an execution failure, from which core dump analysis techniques can only infer partial control and data flows to program crashes. With in-depth analysis on crashed executions, we design MemRay to satisfy more requirements.

For diagnosis techniques via execution analysis, we select two binary-level techniques, Ravel [7] and Layout [45], as baseline techniques. Ravel [7] identifies invalid def-use pairs of instructions. Layout [45] first recovers memory regions, and then identifies different memory regions as vulnerability candidates by comparing passed and failed executions. We exclude AURORA [3] from baseline techniques as AURORA [3] is a statistical technique that requires a large amount of failed executions.

5.2 Experimental Results

Table 1 summarizes our experimental results. Columns 2-4 present vulnerability details, including CVE-ID, type of memory corruption, and root cause. Column # *Inst* shows the number of instructions in dynamic execution traces. The execution trace consists of all traversed instructions belonging to the target program and invoked library functions. Such fine-grained trace enables MemRay to capture memory operations within a library function.

In Column *Time*, the heading *Online* shows the time for generating traces in dynamic execution monitoring, and *Offline* presents the time for off-line vulnerability diagnosis on traces. We observe that the expensive multi-tag taint analysis becomes the performance bottleneck of MemRay. However, considering that MemRay frees security experts from the burden of manually diagnosing vulnerability in tediously long traces, MemRay's overhead is acceptable.

Column MemRay lists the number of invalid data structure references. The three sub-columns, Column `w/o EC`, Column `w EC`, and Column `Root` present the number of invalid data structure references identified in crashes, the number of invalid data structure references identified by execution comparisons, and whether the root cause is identified, respectively. We can observe that the number of invalid data structure references is substantially smaller than the number of executed instructions by several orders of magnitude. Moreover, we observe that the numbers in Column `w/o EC` are much larger than those in Column `w EC`. This result indicates that MemRay encounters several false positives. With execution comparison, our approach can then reduce them. Column # *Size* shows the sizes of invalid data structures. Please note that we denote the size as 0 for dangling pointer dereference.

Column *Ravel [7]* and Column *Layout [45]* present the numbers of identified mismatch data dependencies # *Mismatch* caused by memory corruption respectively.

The most important take-away of Table 1 is MemRay delivers concise results and greatly reduces the amount of effort required for security analysts. Among all of 35 applications, MemRay successfully locates the root cause for 34 of them. The single false negative case comes from `httpd-1.3.35`. Upon further investigation, we find the vulnerable data structure of *CVE-2006-3747* is tampered with a pointer, instead of the program input. We miss the root cause of this vulnerability because we only trace explicit data flow dependencies.

For the false positives listed in Column `w/o EC`, we find that they are mainly caused by imprecise reverse engineered data structures, especially for complicated data structures such as `struct` and C++ objects. In general, the basic insight to reverse engineer program data structures is the memory access patterns reveal types of data structures. However, as demonstrated in Section 4.6, special programming logic or compiler optimization may introduce an inconsistency in these patterns. With execution comparison, we observe that false positive can be counteracted, as indicated in Column `w EC`.

Ravel [7] and Layout [45] report a larger number of mismatches. To be more specific, triggering a memory corruption vulnerability could corrupt multiple neighboring memory cells (such as user-defined variables, metadata in stack or heap). Then, all the reference to these corrupted memory cells will be identified as a mismatch by Ravel [7]. Suspicious memory layouts identified in Layout [45] are specific to program inputs. For different fields in the passed and failed

TABLE 1
Experimental Result Summary

Program	Vulnerability			# Inst	Time(s)		MemRay				Ravel [7]	Layout [45]
	CVE-ID	Type	Root Cause		Online	Offline	w/o EC	w EC	Root	Size	# Mismatch	# Mismatch
streamripper-1.61.25	2006-3124	Stack	mov %al,(%esi, %edx, 1)	781,146	147	37	6	1	✓	50	6	4
newspost-2.1	2005-0101	Stack	`call read`	791,056	141	38	1	1	✓	1024	6	8
mcrypt-2.5.8	2012-4409	Stack	`call fread`	384,620	99	22	2	2	✓	101	4	10
tiffsplit-3.8.2	2006-2656	Stack	`call strcpy`	421,884	108	29	2	2	✓	1024	6	10
gif2png-2.5.2	2009-5018	Stack	`call strcpy`	553,962	117	33	3	3	✓	1032	7	12
ringtonetools-2.22	2004-1292	Stack	mov %dl,−0x404(%ebp, %eax, 1)	896,849	179	50	2	2	✓	1024	5	7
Unalz-0.52	2005-3862	Stack	`call fread`	317,260	81	19	8	2	✓	1560	1	4
O3read-0.03	2004-1288	Stack	mov %bl, 0x2c(%esp, %eax, 1)	333,741	79	19	1	1	✓	1024	4	8
binutils-2.12	2005-4807	Stack	`call vsprintf`	459,127	103	23	1	1	✓	2000	6	14
conquest-8.2a	2007-1371	Stack	mov %al,0x3c(%esp, %ebp, 1)	53,770,057	1537	501	4	3	✓	1024	2	2
poppler-0.24.1	2013-4473	Stack	`call sprintf`	9,910,230	444	166	2	2	✓	1024	3	6
libsmi-0.4.8	2010-2891	Stack	mov %eax, (%edi, %esi, 4)	999,439	178	56	7	3	✓	128	10	12
libpng-1.2.5	2004-0597	Stack	`call fread`	361,625	69	20	8	3	✓	256	4	4
ettercap-0.7.5.1	2013-0722	Stack	`call fscanf`	6,254,698	405	174	1	1	✓	48	3	4
prozilla-1.3.6	2004-1120	Stack	`call sprintf`	1,072,752	189	46	9	2	✓	2048	6	10
mutt-1.4.2.2	2007-2683	Stack	`call memmove`	2,578,343	312	98	2	2	✓	256	10	12
inetutils-1.8	2011-4862	Stack	mov %dl,−0x1(%ebx)	82,419	37	16	11	3	✓	64	5	6
opendchub	2010-1147	Stack	`call sscanf`	378,296	96	20	2	2	✓	50	2	3
Apache-1.3.31	2004-0940	Stack	mov %dl, (%eax)	9,135,126	438	125	1	1	✓	8192	1	2
ghttpd-1.4.3	2001-0820	Stack	`call vsprintf`	424,945	101	21	2	2	✓	200	6	8
putty-0.66	2016-2563	Stack	call sscanf	597,389	339	117	13	1	✓	40	2	2
proftpd-1.3.3a	2010-4221	Stack	mov %dl,(%eax)	566,314	111	45	7	1	✓	1032	2	2
leptonica-1.70.1	2018-7186	Stack	`call sscanf`	710,112	97	31	1	1	✓	512	7	12
Aircrack-ng 1.2	2014-8322	Stack	`callq read`	278,917	135	117	1	1	✓	1024	5	5
xrdp-0.4.1	2008-5904	Heap	`call memcpy`	411,074	117	21	9	2	✓	3216	1	2
nullhttpd-0.5	2002-1496	Heap	`call recv`	690,601	120	21	1	1	✓	1020	1	3
libsndfile-1.0.25	2015-7805	Heap	`call fread`	422,247	110	27	10	2	✓	12292	1	1
0verkill-0.16	2006-2971	int2buf	xor (, %edi, 4),%eax	553,188	129	40	2	2	✓	256	1	1
nginx-1.4.0	2013-2028	int2buf	`call recv`	173,080	88	17	10	2	✓	4096	5	5
httpdx-1.4.5	2009-4769	Format string	`call printf`	582,188	168	87	3	3	✓	0	1	1
gzip-1.2.4	2001-1228	BSS	`call strcpy`	308,773	135	47	1	1	✓	1056	2	2
tiff-4.03	2013-4232	UAF	mov %dl, (%ecx, %eax, 1)	1,279,544	109	47	1	1	✓	0	1	0
openjpeg-2.1.1	2016-7445	Null pointer	movzbl (%eax), %eax	954,716	163	49	1	1	✓	0	0	1
podofo-0.9.4	2017-5854	Null pointer	mov −8(%esp), %ecx	365,182	132	106	1	1	✓	0	1	1
httpd-1.3.35	2006-3747	Off-by-one	cmp %eax, $5	6,297,809	810	268	0	0	×	0	1	0
ncurses-6.2 [14]	**2021-25794**	Stack	mov %esi,0x0(%ebp, %edi, 8)	759,173	96	34	1	1	✓	32	2	4
mp3gain-1.5.2 [15]	NA	Stack	mov %eax,−0x18(%ebp)	6,581,565	762	208	1	1	✓	22	3	3
jhead-3.00 [16]	NA	Heap	mov %eax,−0x94(%ebp)	253,173	52	66	4	4	✓	136	6	8
size-2.26.1 [17]	NA	Heap	mov %eax,0x1c(%esi)	1,009,181	152	67	12	2	✓	48	10	14

program inputs, all the memory layouts correlated to these different fields will become vulnerability candidates. Although Layout [45] leverages mismatch data dependencies to alleviate false positives, it cannot distinguish the cause and effect in a mismatch. By contrast, MemRay detects invalid pointers by excavating data structures, then it can definitely identify the cause and effect respectively. This result reveals that MemRay performs wider applicability to detect the root cause of the vulnerability.

To facilitate understanding vulnerabilities, MemRay delivers high-level semantic information, such as the size of vulnerable data structures. The sizes are identified via reverse engineering. It is possible that the size could be over-estimated. For example, the size of the vulnerable data structure in gzip-1.2.4 is identified as 1056, whereas it is a buffer with 1024-bytes. The reason is that the neighboring objects are not referenced during the execution, thus our approach fails to identify the root pointers in the neighboring object. Over-estimation brings little negative impact on identifying memory corruption, since the memory corruption has resulted in a crash. However, over-estimation could induce the imprecise for understanding vulnerability.

Effectiveness on Unknown Vulnerabilities. We got 24 crash inputs in total on 4 programs. Then we use MemRay to analyze these crashes. The results show that MemRay can accurately locate the root cause of the vulnerabilities. In addition, MemRay can facilitate the diagnosis of these vulnerabilities by constructing data structure reference sequences. The results are shown in the last four rows of Table 1. We have submitted the proposal for CVE and got a CVE number (CVE-2021-25794) [13] for one of them (ncurses-6.2).

5.3 Case Study

In this section, we use several case studies to present our experience of how we leverage MemRay to diagnose vulnerabilities. Moreover, we demonstrate the effectiveness of MemRay by comparing MemRay with other diagnosis techniques with these cases studies.

As shown in Table 1, MemRay can support multiple types of memory corruption vulnerabilities, including stack overflow, heap overflow, integer-to-buffer overflow, use-after-free, double-free, out-of-bounds read, format string vulnerability, null pointer, and data segment (BSS) overflow. For case studies, we only select five types (stack overflow, heap overflow, integer-to-buffer overflow, use-after-free, out-of-bounds read) of vulnerabilities. We did not select double-free, null pointer, BSS overflow, format string vulnerability, because the diagnosis process of MemRay on them is similar to that on selected cases. To be more specific, MemRay diagnoses double-free and null pointer vulnerabilities by checking the alive status of data structures, which is similar to diagnosing use-after-free. Like stack overflow vulnerabilities, MemRay can diagnose BSS overflow by checking whether the reference exceeds the size of data structures. Besides, MemRay diagnoses format string vulnerabilities by checking corrupted stack that is impacted by the format string.

Case Study: Integer-to-Buffer-Overflow in Nginx. The vulnerability *CVE-2013-2028* is a type of integer-to-buffer-overflow [53]. A signed and negative integer is mistakenly used as a parameter for calling `recv`. MemRay first identifies the root cause as `call recv`. As all the parameters of `recv` could lead to memory corruption, the next step is to diagnose which one is the real cause. Further analysis confirms that it is the integer leading to the memory corruption, because the integer propagates from program inputs and MemRay identifies the data structure reference to manipulate this integer in `recv`.

It is difficult to diagnose this vulnerability with reverse execution approaches such as POMP [50], because the execution trace from the root cause to the crash point is long. On this case, POMP [50] records 4,604 instructions as critical. Although these critical instructions contain the root cause, it is still challenging to localize instructions corresponding to the root cause. Ravel [7] identifies some invalid def-use pairs as root cause candidates, but it cannot determine whether def or use is the root cause.

Case Study: Stack Overflow in Mcrypt. The vulnerability *CVE-2012-4409* is a type of classical stack buffer overflow. This program first invokes `fread` to read a single byte of the file. After being converted to an integer, it is treated as the parameter for calling `fread` again, which indicates the buffer size. Therefore, memory corruption occurs when this input byte is too large. MemRay successfully locates the root cause of the vulnerability in the latter `fread`. Afterward, we find that the parameter representing the buffer size is also determined by inputs. A subsequent analysis identifies that this integer is derived from the former `fread`.

On this case, Ravel [7] can locate the vulnerability in the latter `fread`, but ignores the former `fread`. This vulnerability is triggered by the specific value of the data structure rather than the data structure reference itself. Therefore, the def-use pairs exist both in passed and failed executions, and Ravel [7] cannot identify any mismatches. Layout [45] diagnoses vulnerabilities via memory layout recovering and comparison. However, the memory layout on this case is specific to executions, because the buffer size is determined by inputs. Although Layout [45] identifies 10 mismatches by memory layout comparison, it can not furtherly locate the root cause and corresponding instructions.

Case Study: Heap Overflow in Libsndfile. To diagnose the vulnerability *CVE-2015-7805*, MemRay first identifies several candidates for invalid data structure references, and some of them are false positives. With execution comparison, MemRay locates an invalid data structure reference in the invocation of `memcpy`, and the invalid pointer comes from the parameter of the source pointer for calling `memcpy`. That is, there exists an out-of-bounds read `memcpy`. Further, we backtrack the propagation of the source pointers via data dependencies and find that the source pointer is calculated by adding the base address of a data structure in the heap by an integer that propagates from program inputs. Moreover, this integer is overwritten by the invocation of `fread`. In this way, we construct the data dependencies between two memory corruption points and finally identify that the invocation of `fread` overwrites an integer, and this integer is further used to calculate the address of the source memory for invoking `memcpy`. Then, the source pointer becomes invalid and results in a crash.

This vulnerability is triggered only if inputs satisfy a specific branch. AURORA [3] assigns a high score for all instructions on the specific path, and cannot distinguish which instructions correspond to the root cause. POMP [50] records 2,426 instructions as critical because there are frequent memory accesses in the execution. However, it is unable to localize instructions corresponding to the root cause any further.

Case Study: Use-After-Free in Tiff. The vulnerability *CVE-2013-4232* is a type of use-after-free (UAF). By checking the alive status of data structures, MemRay accurately identifies this invalid data structure reference. After that, the key to locating the root cause of UAF errors is to figure out why the program frees the object. By analyzing the data structure reference sequence of this program, MemRay first locates the data structure in the heap which has been freed and change its alive status from true to false. Therefore, the pointer to the freed heap object becomes a dangling pointer, which should not be alive in the following execution. However, this pointer is dereferenced again, leading to the execution of arbitrary code via input bytes.

On this case, POMP [50] records 1,169 critical instructions and manual analysis is still required to locate the root cause. Layout [45] diagnoses vulnerabilities by identifying different memory layouts in passed and failed executions. However, use-after-free belongs to temporal memory errors which do not change memory layouts. Therefore, Layout [45] cannot identify this kind of vulnerabilities.

Case Study: Out-of-Bounds Read in Ncurses. We further use MemRay to analyze some unknown vulnerabilities and finally receive a CVE number (CVE-2021-25794) [13]. There is a 4-byte out-of-bounds read on this case. Later, this 4-bytes item is passed to `strlen` as a pointer in the following execution, and then results in a crash. MemRay backtracks the taint tags and finds that this program repeatedly reads 4-byte data and processes the data as a pointer in a loop. Furthermore, MemRay diagnoses that a certain byte in the input determines the number of loops.

6 Related Work

Source-Code Level Techniques. To automate vulnerability diagnosis, researchers have proposed multiple techniques, such as memory error debugging [32], [38], [51] and memory corruption detection [1], [2], [6]. However, most of these techniques require the support of source code. This limits their adoptions when the source code is not available, such as analyzing vulnerabilities in commercial software.

Core Dump Analysis Techniques. Core dump analysis techniques, including POMP [50], POMP++ [31], REPT [10], and kernel REPT [22], reverse execute the program from the crash point, and then locate critical instructions leading to the crash. These approaches aim at causal analysis for crashes in a deployed system by reconstructing the data flow and historical states.

As a core dump only provides a memory snapshot of an execution failure, from which core dump analysis techniques can only infer partial control and data flows pertaining to program crashes. Therefore, these techniques are generally unable to locate the root cause of the vulnerability precisely. By contrast, MemRay aims to diagnose vulnerabilities via in-depth analysis with the support of a crashed execution. Thus, we design MemRay not only to locate root causes but also to provide explanations with data structure reference sequence about the vulnerability behind the crash, which can help security analysts to understand the vulnerabilities.

Vulnerability Diagnosis via Execution Analysis. MemRay diagnoses vulnerabilities via in-depth analysis on execution traces. These types of techniques, including Ravel [7], AURORA [3], and Layout [45], typically instrument a program so that one can log program states (i.e., the input to a program as well as the memory accesses). Later, analysts can utilize the log to replay the program and perform in-depth analysis. With such in-depth analysis, analysts can reconstruct the control and data flows. Further, they can reason crashes with these control and data flows.

As execution traces can deliver more detailed information, such types of techniques are often designed not only to locate the root cause of vulnerabilities but also to provide some semantic information for facilitating understanding vulnerabilities, especially at the binary level where high-level program abstractions are missing.

Ravel [7] introduces def-use pairs to locate critical instructions related to the crash. It computes def-use pairs in passed executions and then identifies illegal def-use pairs if they only exist in a failed execution. In this way, it can identify invalid data dependencies. However, Ravel [7] requires distinguishing which one (the def or the use) is the root cause for a suspicious def-use pair. Although Ravel [7] defines some heuristics to address this challenge, these heuristics may not complete and precious. In addition, root causes identified by Ravel [7] are instructions at the binary level. Compared with Ravel [7], MemRay can precisely locate the root cause and distinguish the memory access type (read or write) with the help of data structure reference sequences.

Layout [45] recovers memory layouts from both passed and failed executions, and then identifies suspicious memory layouts that only exist in failed executions as vulnerability candidates. Both Layout [45] and MemRay aim to diagnose root causes by providing high-level semantics beyond instructions. The main limitation is memory layouts excavated in Layout [45] are specific to executions. As the passed and failed executions can be entirely different, there may be a large number of mismatches in comparison, which may introduce large false positives. Although Layout [45] leverages mismatch data dependencies to alleviate false positives, it cannot distinguish the cause and effect in a mismatch. Compared with Layout [45], MemRay aims at recovering data structures rather than specific memory layouts, which can be more precise and allow for a better description of the vulnerability.

AURORA [3] introduces a statistical approach called predicated-based root cause analysis. Through statistical analysis of the execution status (memory and registers), AURORA [3] generates a lot of predicates on each instruction. Then it selects the predicates which have a higher correct rate and considers the corresponding instructions as the root cause. However, AURORA [3] may locate both the root cause and the error state caused by the vulnerability due to error state propagation, as demonstrated in previous work [55]. Compared with AURORA [3], MemRay identifies memory corruption by detecting violations in the input manipulations via data structures. In this way, MemRay can precisely locate the root cause of vulnerabilities.

7 Discussion

Our approach cannot deal with memory corruption in interpreters caused by interpretative programming languages, such as `PHP` and `JavaScript`. In such cases, program inputs of interpreters, including both data and code, will be tainted and our dynamic monitor cannot distinguish them.

MemRay encounters a false negative on *CVE-2006-3747*, which is caused by pointer propagation instead of input pointers. As the pointer does not directly depend on program input, we missed this memory corruption. Besides, our approach may also encounter false negatives caused by under-tainting problem, which due to control flow dependency is a common limitation of dynamic taint analysis [37]. DTA++ [27] provides mitigation to implicit taint propagation. However, as demonstrated in the literature [27], DTA+ + may also introduce the over-tainting problem that can cause more false positives.

8 Conclusion

In this paper, we propose a novel dynamic approach to memory-corruption vulnerability diagnosis, called MemRay. MemRay can characterize the invalid data structure references to manipulate inputs caused by memory corruption vulnerabilities. Besides, it delivers rich semantics to locate root cause and assist in understanding vulnerabilities. Evaluations on real-world vulnerabilities demonstrate the efficacy of our approach.

References

[1] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, “Preventing memory error exploits with WIT,” in *Proc. IEEE Symp. Secur. Privacy*, 2008, pp. 263–277.

[2] P. Akritidis, M. Costa, M. Castro, and S. Hand, “Baggy Bounds Checking: An efficient and backwards-compatible defense against out-of-bounds errors,” in *Proc. 18th Conf. USENIX Secur. Symp.*, 2009, pp. 51–66.

[3] T. Blazytko *et al.*, “AURORA: Statistical crash analysis for automated root cause explanation,” in *Proc. 29th USENIX Secur. Symp.*, 2020, Art. no. 14.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, “Towards automatic generation of vulnerability-based signatures,” in *Proc. IEEE Symp. Secur. Privacy*, 2006, pp. 15–16.

[5] J. Caballero, H. Yin, Z. Liang, and D. Song, “Polyglot: Automatic extraction of protocol message format using dynamic binary analysis,” in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 317–329.

[6] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in *Proc. 7th Symp. Oper. Syst. Des. Implementation*, 2006, pp. 147–160.

[7] Y. Chen, M. Khandaker, and Z. Wang, “Pinpointing vulnerabilities,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 334–345.

[8] W. D. Clinger, “Proper tail recursion and space efficiency,” in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1998, pp. 174–185.

[9] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, “Bouncer: Securing software by blocking bad input,” in *Proc. 21st ACM SIGOPS Symp. Oper. Syst. Princ.*, 2007, Art. no. 1.

[10] W. Cui *et al.*, “REPT: Reverse debugging of failures in deployed software,” in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implementation*, 2018, pp. 17–32.

[11] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, “RETracer: Triaging crashes by reverse execution from partial memory dumps,” in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 820–831.

[12] CVE-2001–0820, 2001. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001–0820

[13] CVE-2021–25794, 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021–25794

[14] ncurses, 2020. [Online]. Available: http://invisible-island.net/ncurses/

[15] mp3gain, 2010. [Online]. Available: http://mp3gain.sourceforge.net/

[16] jhead, 2015. [Online]. Available: http://www.sentex.net/ mwandel/jhead/

[17] binutils, 2016. [Online]. Available: http://www.gnu.org/software/binutils/

[18] K. A. Farris, A. Shah, G. Cybenko, R. Ganesan, and S. Jajodia, “VULCON: A system for vulnerability prioritization, mitigation, and management,” *ACM Trans. Privacy Secur.*, vol. 21, no. 4, Jun. 2018, Art. no. 16.

[19] S. Gan *et al.*, “GREYONE: Data flow sensitive fuzzing,” in *Proc. 29th USENIX Secur. Symp.*, 2020, Art. no. 145.

[20] S. Gan *et al.*, “CollAFL: Path sensitive fuzzing,” in *Proc. 39th IEEE Symp. Secur. Privacy*, 2018, pp. 679–696.

[21] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, “Beyond tests: Program vulnerability repair via crash constraint extraction,” *ACM Trans. Softw. Eng. Methodol.*, vol. 30, 2021, Art. no. 14.

[22] X. Ge, B. Niu, and W. Cui, “Reverse debugging of kernel failures in deployed systems,” in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 281–292.

[23] C. L. Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[24] A. Henderson *et al.*, “Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 248–258.

[25] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and L. Wai-Meng, “CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations,” in *Proc. IEEE 6th Int. Conf. Softw. Secur. Rel.*, 2012, pp. 78–87.

[26] Z. Huang, D. Lie, G. Tan, and T. Jaeger, “Using safety properties to generate vulnerability patches,” in *Proc. IEEE Symp. Security Privacy*, 2019, pp. 539–554.

[27] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, “DTA++: Dynamic taint analysis with targeted control-flow propagation,” in *Proc. 18th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2011.

[28] Z. Li *et al.*, “VulDeePecker: A deep learning-based system for vulnerability detection,” in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[29] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, “Sound input filter generation for integer overflow errors,” in *Proc. 41st ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2014, pp. 439–452.

[30] D. Mu *et al.*, “Understanding the reproducibility of crowd-reported security vulnerabilities,” in *Proc. 27th USENIX Conf. Secur. Symp.*, 2018, pp. 919–936.

[31] D. Mu *et al.*, “POMP++: Facilitating postmortem program diagnosis with value-set analysis,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1929–1942, Sep. 2021.

[32] S. Pearson *et al.*, “Evaluating and improving fault localization,” in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 609–620.

[33] H. Peng, Y. Shoshitaishvili, and M. Payer, “T-Fuzz: Fuzzing by program transformation,” in *Proc. 39th IEEE Symp. Secur. Privacy*, 2018, pp. 697–710.

[34] G. Portokalidis, A. Slowinska, and H. Bos, “Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation,” in *Proc. 1st ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2006, pp. 15–27.

[35] A. Prakash, H. Yin, and Z. Liang, “Enforcing system-wide control flow integrity for exploit detection and diagnosis,” in *Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Secur.*, 2013, pp. 311–322.

[36] A. Ramaswamy, S. Bratus, S. W. Smith, and M. E. Locasto, “Katana: A hot patching framework for ELF executables,” in *Proc. Int. Conf. Availability Rel. Secur.*, 2010, pp. 507–512.

[37] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 317–331.

[38] E. C. Sezer, P. Ning, C. Kil, and J. Xu, “Memsherlock: An automated debugger for unknown memory corruption vulnerabilities,” in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 562–572.

[39] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury, “Localizing vulnerabilities statistically from one exploit,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2021, pp. 537–549.

[40] A. Slowinska, T. Stancescu, and H. Bos, “Howard: A dynamic excavator for reverse engineering data structures,” in *Proc. 18th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2011.

[41] A. Sotirov, “Hotpatching and the rise of third-party patches,” in *Proc. Black Hat Technical Secur. Conf., Las Vegas, Nevada*, 2006, vol. 83, p. 88.

[42] N. Stephens *et al.*, “Driller: Augmenting fuzzing through selective symbolic execution,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–16.

[43] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal war in memory,” in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 48–62.

[44] V. V. D. Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos, “Memory errors: The past, the present, and the future,” in *Proc. 15th Int. Conf. Res. Attacks Intrusions Defenses*, 2012, pp. 86–106.

[45] H. Wang *et al.*, “Locating vulnerabilities in binaries via memory layout recovering,” in *Proc. 27th ACM Joint Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.*, 2019, pp. 718–728.

[46] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, “Detecting “0-day” vulnerability: An empirical study of secret security patch in OSS,” in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2019, pp. 485–492.

[47] Y. Wang *et al.*, “Revery: From proof-of-concept to exploitable,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1914–1927.

[48] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, “CrashLocator: Locating crashing faults based on crash stacks,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 204–214.

[49] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, “CREDAL: Towards locating a memory corruption vulnerability with your core dump,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 529–540.

[50] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, “POMP: Postmortem program analysis with hardware-enhanced post-crash artifacts,” in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 17–32.

[51] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, “Automatic diagnosis and response to memory corruption vulnerabilities,” in *Proc. 12th ACM Conf. Comput. Commun. Secur.*, 2005, pp. 223–234.

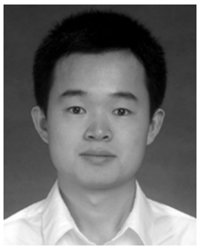
[52] T. Yue *et al.*, “EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit,” in *Proc. 29th USENIX Secur. Symp.*, 2020, Art. no. 130.

[53] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, “IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time,” in *Proc. Eur. Symp. Res. Comput. Secur.*, 2010, pp. 71–86.

[54] M. Zhang, A. Prakash, X. Li, Z. Liang, and H. Yin, “Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis,” in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012.

[55] Z. Zhang, W. K. Chan, T. Tse, B. Jiang, and X. Wang, “Capturing propagation of infected program states,” in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf.* and *ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 43–52.

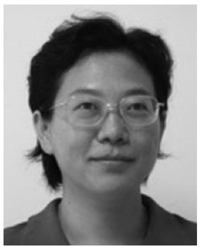
[56] L. Zhao, Y. Duan, H. Yin, and J. Xuan, “Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/NDSS2019_Proceedings_Front_Matter.pdf



Lei Zhao received the BSc and PhD degrees in computer science and engineering both from Wuhan University, China, in 2007 and 2012, respectively. He is currently a professor with the School of Cyber Science and Engineering, Wuhan University, China. His research interests include software testing and debugging, program analysis, and software security.

Keyang Jiang received the BSc degree in computer science and engineering from Wuhan University, China, in 2019. He is currently working toward the graduate degree with the School of Cyber Science and Engineering, Wuhan University, China. His research interests include binary code analysis and symbolic execution.

Yuncong Zhu received the BSc degree in computer science and engineering from Wuhan University, China, in 2019. He is currently working toward the graduate degree with the School of Cyber Science and Engineering, Wuhan University, China. His research interests include program analysis and software security.



Lina Wang is currently a professor with the School of Cyber Science and Engineering, Wuhan University, China. Her research interests include system security, network security, and AI security.

Jiang Ming received the PhD degree from the College of Information Sciences and Technology, Penn State University, State College, Pennsylvania, in 2016. He is currently an assistant professor with the Computer Science and Engineering Department, UT Arlington, Arlington, Texas. His research interests include binary code analysis and verification for security issues, hardware-assisted malware analysis, and mobile systems security.