POMP++: Facilitating Postmortem Program Diagnosis with Value-set Analysis

Dongliang Mu, *Member, IEEE,* Yunlan Du, Jianhao Xu, Jun Xu, *Member, IEEE,* Xinyu Xing, *Member, IEEE,* Bing Mao, *Member, IEEE,* and Peng Liu, *Member, IEEE*

Abstract—With the emergence of hardware-assisted processor tracing, execution traces can be logged with lower runtime overhead and integrated into the *core dump*. In comparison with an ordinary core dump, such a new post-crash artifact provides software developers and security analysts with more clues to a program crash. However, existing works only rely on the resolved runtime information, which leads to the limitation in data flow recovery within long execution traces.

In this work, we propose POMP++, an automated tool to facilitate the analysis of post-crash artifacts. More specifically, POMP++ introduces a reverse execution mechanism to construct the data flow that a program followed prior to its crash. Furthermore, POMP++ utilizes Value-set Analysis, which helps to verify memory alias relation, to improve the ability of data flow recovery. With the restored data flow, POMP++ then performs backward taint analysis and highlights program statements that actually contribute to the crash.

We have implemented POMP++ for Linux system on x86-32 platform, and tested it against various crashes resulting from 31 distinct real-world security vulnerabilities. The evaluation shows that, our work can pinpoint the root causes in 29 cases, increase the number of recovered memory addresses by 12% and reduce the execution time by 60% compared with existing reverse execution. In short, POMP++ can accurately and efficiently pinpoint program statements that truly contribute to the crashes, making failure diagnosis significantly convenient.

Index Terms—Postmortem Program Diagnosis, Failure Diagnosis, Reverse Execution, Value-set Analysis.

1 Introduction

Due to the increasing complexity of functionality, software systems inevitably contain defects despite developers' best efforts. When these defects are triggered, a program typically crashes and terminates abnormally. Vendors of modern software systems can receive millions of crash reports every day [2, 3]. This practice highly motivates the techniques of *efficient* and *accurate* postmortem program diagnosis.

Briefly speaking, postmortem program diagnosis is to identify the program statements pertaining to the crash, analyze these statements, and eventually figure out why a bad value was passed to the crash site. Of all the techniques in postmortem program diagnosis, record-and-replay [4–6] and core dump analysis [2, 7, 8] are the most effective solutions but still suffer from their essential limitations. Compared with record-and-replay which requires high-overhead program instrumentation, core dump analysis is more lightweight and promising. However, the core dump provides only a snapshot of the failure, from which people can only infer partial control and data flows pertaining to program crashes.

Recently, advances in hardware-assisted processor tracing significantly have ameliorated this situation. With the emergence of Intel PT [9] – a new hardware feature in Intel CPUs – software developers and security analysts can trace executed instructions and save them in a circular buffer. When a crash occurs, the operating system includes the trace into a core dump. Since this post-crash artifact contains both the state of crashing memory and the execution history, software developers not only can inspect the program state at the time of the crash, but also fully reconstruct the control flow that led to the crash, making software debugging more informative and efficient. However, it is still time and resource consuming to diagnose the root causes of software failures with PT trace, because such a post-craft artifact typically contains too many instructions to be examined manually.

To address this problem, many solutions have been proposed [1, 10]. Our earlier version POMP [1] is an automatic root cause analysis tool, which reconstructs a data flow based on PT trace and coredump. During data flow recovery, when there is a need to resolve memory aliases, it recursively runs hypothesis testing (HT) which makes assumptions about the alias relations and reject the wrong assumptions according to run-time information. Another recently proposed technique REPT [10] reconstructs the execution states by combining online PT tracing and offline binary analysis. It performs forward/backward analysis iteratively based on memory dump and applies an error correction scheme to resolve conflicts during reverse execution.

Nevertheless, we recognize that the existing works share common shortcomings that limit their use in practice. On the one hand, most of the previous works only rely on run-time information. Once such information is not sufficiently recovered, the postmortem analysis may not proceed further and miss the root cause of crashes. On the other hand, the existing works have limited analysis efficiency. For instance, POMP handles missing memory writes by running hypothesis tests recursively, incurring exponential computation complexity. This low efficiency makes POMP impractical to handle millions of crash reports on a daily

D. Mu, Y. Du, Jianhao Xu and B. Mao are with Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China (email: dzm77@ist.psu.edu; {duyunlan,jianhao_xu}@smail.nju.edu.cn; maobing@nju.edu.cn).
Jun Xu is with Department of Computer Science, Stevens institute of technology, Hoboken, NJ 07030, USA (e-mail: jxu69@stevens.edu).
X. Xing, and P. Liu are with College of Information Sciences and Technology, the Pennsylvania State University, Pennsylvania 16802, USA (e-mail: {xxing, pliu}@ist.psu.edu).
An earlier version appeared at the 26th USENIX Security Symposium (USENIX Security '17) [1].

basis.

In this work, we propose POMP++, a system to address the above limitations in analyzing a post-crash artifact and pinpointing statements pertaining to the crash. We augment POMP with Value-set Analysis (VSA) [11], the most effective and efficient alias analysis at the binary code level [12]. The intuition behind is that static analysis can facilitate the understanding of alias relations. Technically speaking, we pre-perform a customized version of VSA to obtain the address set of each memory access. When encountering an uncertain alias during reverse execution, we query VSA results for answers. Accordingly, we introduce the improving scheme of reverse execution based on VSA and hypothesis testing, detailed in Section 4. With the VSA-enhanced data flow recovery, we finally utilize backward taint analysis to pinpoint the critical instructions leading up to the crash.

The novelty of this work lies in two aspects. First, we propose a new VSA-based approach for memory alias verification. As the original VSA is designed to analyze an entire binary, we customize the VSA algorithms to support the straight-line trace carried by a post-crash artifact. In particular, we introduce new schemes to identify memory regions and we develop solutions to handle traces that have incomplete contexts. More details are presented in Section 4. Second, we develop new schemes to incorporate our customized VSA to POMP. The goal is to achieve bi-directional feedback so that VSA gets improved with information recovered by reverse execution and in turn aids the alias verification of reverse execution. Accordingly, we introduce two new hybrid schemes. As we will demonstrate in Section 6, our design with the hybrid schemes significantly improves the recovery of data flow, leading to better root cause identification. Moreover, as a large amount of hypothesis tests are replaced by VSA queries, POMP++ achieves superior efficiency compared with POMP.

In summary, this paper makes the following contributions.

- We design POMP++, a new technique that analyzes post-crash artifacts by reversely executing instructions residing in the post-crash artifact.
- We develop a VSA-enhanced reverse execution scheme to enable almost constant utilities especially when reversely executing long instruction trace.
- We implement POMP++ in 32-bit Linux to facilitate the job of software developers and security analysts when pinpointing software defects.
- We evaluate POMP++ on 31 distinct real-world security vulnerabilities, and compare with POMP. With better identification of root causes, 12% of data flow recovery improvement, and 60% of efficiency improvement, we demonstrate that VSA can facilitate postmortem program diagnosis.

The rest of this paper is organized as follows. Section 2 describes the threat model of our research. Section 3 presents a high-level work flow of POMP++. Section 4 and 5 describe the design and implementation of POMP++ in detail. Section 6 evaluates the utility of POMP++. Section 7 summarizes the related work followed by some discussions on POMP++ in Section 8. Finally, we conclude this work in Section 9.

Compared with the earlier version [1], our work has many differences in almost all sections. With static analysis, we additionally work on how to extend utilities of reverse execution while improving the time efficiency in the context of postmortem program diagnosis. Note that the time efficiency really matters when faced with tremendous amounts of software crashes. Thus, we propose our new approaches with VSA, which is described briefly in Section 3 and detailed its customization, calculation and application to alias verification in Section 4. We then correspondingly introduce its extra implementation in Section 5 and make comparison with the earlier version [1] in Section 6.

```
typedef struct A {
   int t;
   void (*func)(void);
}SA;

void test(){
   ......
}

int child(int *a) {
   a[0] = a;
   a[1] = 0x0;
   return 0;
}

int main() {
   SA *p = (SA *)malloc(sizeof(SA));
   p->func = test;
   child(&(p->t));
   (p->func)(); // crash site
}
```

Fig. 1. A toy example with a heap overflow defect.

2 THREAT MODEL

In this work, we focus on diagnosing the crash of a process. As a result, we exclude the program crashes that do not incur the unexpected termination of a running process (*e.g.,* Java program crashes). Since this work diagnoses a process crash by analyzing a post-crash artifact, we further exclude those process crashes that typically do not produce an artifact.

A post-crash artifact contains not only the memory snapshot of a crashing program but also the instructions that the program followed prior to its crash[1]. Aiming to determine instructions that actually pertain to the crash, we assume a post-crash artifact carries all the instructions that actually contribute to the crash. We believe this is a realistic assumption because a software defect is typically close to a crash site [13–15] and the operating system can easily allocate a memory cell to store the execution trace from a defect triggered to an actual crash. It should be noted that we do not assume the source code of the crashing program is available.

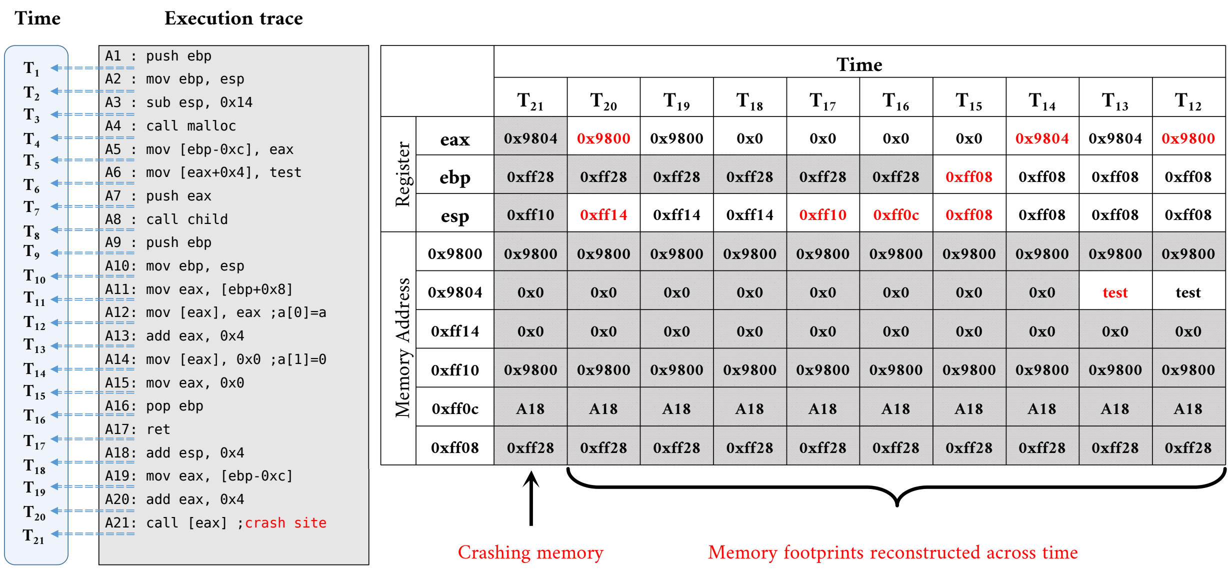
3 DESIGN GOAL AND APPROACHES

In this section, we describe the objective of our research. We then give a demonstrative example to illustrate our basic idea and high-level work flow of how POMP++ performs reverse execution, Value-set Analysis and root cause pinpointing.

3.1 Objective

The goal of software failure diagnosis is to identify the root cause of a failure from the instructions enclosed in an execution trace. Due to the enormous number of instructions in the trace, software developers' digging for the root cause is tough and time-consuming.

1. While Intel PT does not log unconditional jumps and linear code, a full execution trace can be easily reconstructed from the execution trace enclosed in a post-crash artifact. By an execution trace in a post-crash artifact, without further specification, we mean a trace including conditional branch, unconditional jump and linear code.



```
A1 : push ebp
A2 : mov ebp, esp
A3 : sub esp, 0x14
A4 : call malloc
A5 : mov [ebp-0xc], eax
A6 : mov [eax+0x4], test
A7 : push eax
A8 : call child
A9 : push ebp
A10: mov ebp, esp
A11: mov eax, [ebp+0x8]
A12: mov [eax], eax ;a[0]=a
A13: add eax, 0x4
A14: mov [eax], 0x0 ;a[1]=0
A15: mov eax, 0x0
A16: pop ebp
A17: ret
A18: add esp, 0x4
A19: mov eax, [ebp-0xc]
A20: add eax, 0x4
A21: call [eax] ;crash site
```

		Time									
		T_{21}	T_{20}	T_{19}	T_{18}	T_{17}	T_{16}	T_{15}	T_{14}	T_{13}	T_{12}
Register	eax	0x9804	0x9800	0x9800	0x0	0x0	0x0	0x0	0x9804	0x9804	0x9800
	ebp	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff08	0xff08	0xff08	0xff08
	esp	0xff10	0xff14	0xff14	0xff14	0xff10	0xff0c	0xff08	0xff08	0xff08	0xff08
Memory Address	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800
	0x9804	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	test	test
	0xff14	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
	0xff10	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800	0x9800
	0xff0c	A18	A18	A18	A18	A18	A18	A18	A18	A18	A18
	0xff08	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28

Fig. 2. A post-crash artifact along with the execution state recovered by reversely executing the trace enclosed in the artifact. Note that, for simplicity, all the memory addresses and the value in registers are trimmed and represented with two hex digits. Note that `A18` and `test` indicate the addresses at which the instruction and function are stored.

With our VSA-enhanced reverse execution technique, we want to pinpoint the minimal set of instructions that contribute to the crash, making postmortem program analysis significantly easier.

3.2 Reverse Execution

With Intel PT, the instruction trace prior to the crash is available for reverse execution. To illustrate the method and challenges during the reverse execution, we take the code in Figure 1 as an example to show the process of recovering the execution state with the recorded execution trace.

The program in Figure 1 crashes at `line 20`, caused by an overflow that occurs at `line 12`. After the crash, an execution trace is left behind in a post-crash artifact shown in Figure 2. In addition to the trace, the artifact captures the state of the crashing memory which is illustrated as the values shown in column T_{21}.

When a software developer or security analyst begins to follow through the execution trace reversely from the crash, he will be prematurely blocked at instruction `A19` because `mov` overwrites register `eax` and an inverse operation against such instruction lacks information to restore its previous value. To solve this problem and continue backtracking, we construct a data flow based on a use-define chain to perform forward analysis. We will detail the forward analysis in Section 4. Then, we can easily observe that instruction `A15` specifies the definition of register `eax`, and that definition can reach instruction `A19` without any other intervening definitions. As a result, we can restore the value in register `eax` and thus complete the inverse operation for instruction `A19`.

For arithmetical instructions like `A18`, it is easy to recover the prior memory footprints by doing inverted computation. For complex instructions such as `A17`, we can treat them as `mov eip, [esp]` and then `add esp, 0x4`, and complete the reverse operations. Following these instinctive reversal rules, the reverse execution can further restore memory footprints.

However, when backward analysis reaches instruction `A14`, through forward analysis, we can discover that the value in register `eax` after the execution of `A14` is dependent upon both instruction `A11` and `A13`. As we then need to retrieve the value stored in the memory region specified by `[ebp+0x8]` shown in instruction `A11`, we are confronted with the problem that the memory indicated by `[ebp+0x8]` might be overwritten by `[eax]` in instruction `A12` and/or `[eax]` in instruction `A14`, as they might be different symbolic names that access data in the same memory location (*i. e.,* alias).

To address this issue, `POMP` employs hypothesis testing (HT) to verify possible memory alias relations. To be specific, `POMP` makes two hypotheses, one assuming two symbolic names are aliases of each other while the other assuming the opposite. Then, it tests each of these hypotheses by emulating inverse operations for instructions. Let's continue the example to verify alias relation between `[ebp+0x08]` in `A11` and `[eax]` in `A14`. For the first hypothesis that they are aliases, after the inverse operation for instruction `A15`, the information carried by the memory footprint at T_{14} will have three constraints, including $eax = ebp + 0x8$, $eax = [ebp + 0x8] + 0x4$ and $[eax] = 0$. For the opposite hypothesis, the constraint set will include $eax \neq ebp + 0x8$, $eax = [ebp + 0x8] + 0x4$ and $[eax] = 0$. By looking at the memory footprint at T_{14} and examining these two constraint sets, reverse execution can easily reject the first hypothesis and accept the second because constraint $eax = ebp + 0x8$ for the first hypothesis does not hold.

However, when it comes to the alias relation between `[ebp+0x08]` in `A11` and `[eax]` in `A12`, HT cannot make a conclusion because it does not have sufficient information to find any conflict. In order to solve this problem, we introduce static analysis (*i. e.,* Value-set Analysis) [11] to `POMP`. In the following, we describe how Value-set Analysis helps determine the above alias relation.

3.3 Value-set Analysis for Alias Verification

Over the past decades, there have been many works proposed to perform alias analysis at the binary code level [16–18]. Regarded as the most effective and efficient technique [12], Value-set Analysis purely relies on static information in the binary and such information rarely reduces as the trace becomes longer.

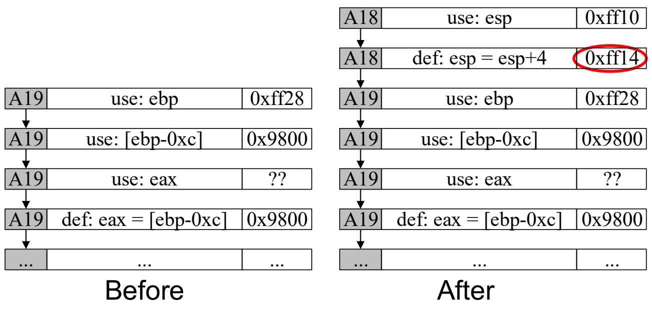

A19 | use: ebp | 0xff28
A19 | use: [ebp-0xc] | 0x9800
A19 | use: eax | ??
A19 | def: eax = [ebp-0xc] | 0x9800
... | ... | ...
Before
A18 | use: esp | 0xff10
A18 | def: esp = esp+4 | 0xff14
A19 | use: ebp | 0xff28
A19 | use: [ebp-0xc] | 0x9800
A19 | use: eax | ??
A19 | def: eax = [ebp-0xc] | 0x9800
... | ... | ...
After

Fig. 3. A use-define chain before and after appending new relations derived from instruction `A18`. Each node is partitioned into three cells. From left to right, the cells carry instruction ID, definition (or use) specification and the value of the variable. Note that symbol ?? indicates the value of that variable is unknown.

Based on the observed patterns that memory layout generally follows, VSA partitions memory into several disjoint memory regions, such as stack, heap, and global, and assigns memory accesses to the regions accordingly. For some memory accesses, VSA achieves region assignment by examining the semantics of the instructions. For others, VSA performs forward data flow analysis to determine the regions conservatively.

Observing the region division, we can easily address the aforementioned alias relation between `[ebp+0x08]` in `A11` and `[eax]` in `A12` in Figure 2. Inferred from the specification of `malloc` function in `A4`, the return value, *i. e.,* `eax` register is a pointer to the allocated heap area. Therefore, the value of eax in `A5` and `A6` are all linked to heap region. Inferred from the semantices of `A5`, [ebp-0xc] points to the heap region. Furthermore, when `[ebp-0xc]` in `A5` is transmitted to `[ebp+0x8]` in `A11` without any intervening definitions, `[eax]` in `A12` points to the heap region as well. Meanwhile, based on the frame pointer `ebp`, `[ebp+0x8]` in `A11` is an access to the stack region, which indicates that it cannot be an alias of `[eax]` in `A12`.

Besides the division in memory region, VSA also calculates an over-approximation of the set of addresses on which memory accesses span. With the results given by VSA, we can narrow down the range of aliases to be verified and significantly facilitate reverse execution. In this way, we design `POMP++` with VSA to assist memory alias verification. The details of VSA computation and `POMP++` will be presented in Section 4.

3.4 Root Cause Pinpointing

After the recovery of memory footprints shown in Figure 2, software developers and security analysts can easily derive the corresponding data flow and thus pinpoint instructions that truly contribute to a crash. In our work, `POMP++` automates this procedure by using backward taint analysis. By examining the memory footprints restored, `POMP++` can easily find out that the memory indicated by `[eax]` in instruction `A21` shares the same address with that indicated by `[eax]` in instruction `A14`. This implies that the bad value is actually propagated from instruction `A14`. As such, `POMP++` highlights instructions `A21` and `A14`, and deems they are truly attributable to the crash. More details about the backward taint analysis are shown in Section 4.

4 Design

In this section, we present the design of `POMP++`. We first describe how we customize Value-set Analysis on the execution trace. Then, we elaborate on the core algorithm to perform reverse execution and memory footprint recovery. Finally, we explain some design details about the backward taint analysis which helps to pinpoint instructions truly contributing to the crash.

4.1 Value-set Analysis Customization

As a static analysis for binary code, VSA can be performed prior to reverse execution. In our problem settings, there are several challenges to maximize the utility of VSA. First, the existing VSA solutions lack schemes for the identification of many memory regions. We develop new schemes understand memory regions according to the patterns of instruction sequences. Second, the original VSA is designed to analyze an entire binary other than a straight-line execution trace. As handling a single trace has much lower complexity and leads to better accuracy, we customize the VSA algorithms to support the straight-line trace carried by a post-crash artifact. Third, we leverage the recovered information from reverse execution to maximize the utility of VSA. In the following, we detail the details of our designs.

4.1.1 Memory Regions Identification

Compilers often generate code that uses different patterns to access different memory regions. This gives us the foundation to infer region information — we seek those patterns from the instruction trace and map them into region information. As we mentioned in Section 3, for **Stack** and **Heap** regions, we rely on the stack frame pointer and heap-based dynamic allocation functions, respectively. In addition to those patterns in original VSA [11], we also summarize some new ones in the single trace as follows.

Global variables. In position dependent code, global variables hard-code their addresses in instructions, which can be easily determined.

```
call get_pc_thunk.cx ;mov PC to ebx
add ebx, 0x11b0 ;get locations of GOT
mov eax, [ebx-0x10] ;get address from entry
mov eax, [eax] ;access global variable
```

In position independent code (PIC) [19], global variables are accessed with reference to Program Counter (PC) and the Global Offset Table (GOT). We use the above example as an illustration. The code first retrieves PC with a special function (`get_pc_thunk.cx`). Then, it adds a constant value to PC, obtaining the location of the GOT. Finally, the last two instructions locate and access the global variable, respectively. This access pattern is generally unique to determine global variables.

Static variables and TLS variables. In position dependent code, static variables also hard-code their addresses in instructions, and TLS variables use a special register(*e.g.,* `gs` in `x86-32`) as base address. Both of these two cases are identifiable without difficulties. In PIC code, static variables are located using PC and TLS variables are accessed with PC, GOT, and the aforementioned special register. These accesses are also separable from accesses to other regions.

Special functions. Going beyond the patterns above, we include an additional strategy to determine three types of functions that only access particular regions:

- Functions that perform no memory modification or only write to its own stack (*e.g.,* `getchar`).
- Functions that only make changes to its own stack and TLS variables (*e.g.,* `lseek`).

TABLE 1
A demonstrative example indicating value-set analysis with respect to its capability of performing alias analysis.

Index	*A-loc*	Value-set
A1	`esp`	(⊥, [-0x4, -0x4], ⊥)
A2	`ebp`	(⊥, [-0x4, -0x4], ⊥)
A3	`esp`	(⊥, [-0x18, -0x18], ⊥)
A5	`[ebp-0xc]`(⊥, [-0x10, -0x10], ⊥)	(⊥, ⊥, [0x0, 0x0])
	`eax`	(⊥, ⊥, [0x0, 0x0])
A6	`[eax+0x4]`(⊥, ⊥, [0x4, 0x4])	([test, test], ⊥, ⊥)
A7	`esp`	(⊥, [-0x1c, -0x1c], ⊥)
	`[esp]`(⊥, [-0x1c, -0x1c]), ⊥)	(⊥, ⊥, [0x0, 0x0])
A8	`esp`	(⊥, [-0x20, -0x20], ⊥)
	`[esp]`(⊥, [-0x20, -0x20], ⊥)	([A18, A18], ⊥, ⊥)
A9	`esp`	(⊥, [-0x24, -0x24], ⊥)
	`[esp]`(⊥, [-0x24, -0x24], ⊥)	(⊥, [-0x4, -0x4], ⊥)
A10	`ebp`	(⊥, [-0x24, -0x24], ⊥)
A11	`[ebp+0x8]`(⊥, [-0x1c, -0x1c], ⊥)	(⊥, ⊥, [0x0, 0x0])
	`eax`	(⊥, ⊥, [0x0, 0x0])
A12	`eax`	(⊥, ⊥, [0x0, 0x0])
	`[eax]`(⊥, ⊥, [0x0, 0x0])	(⊥, ⊥, [0x0, 0x0])
A13	`eax`	(⊥, ⊥, [0x0, 0x0])
A14	`[eax]`(⊥, ⊥, [0x4, 0x4])	([0x0, 0x0], ⊥, ⊥)
A15	`eax`	([0x0, 0x0], ⊥, ⊥)
A16	`ebp`	(⊥, [-0x4, -0x4], ⊥)
	`esp`	(⊥, [-0x20, -0x20], ⊥)
A17	`esp`	(⊥, [-0x1c, -0x1c], ⊥)
A18	`esp`	(⊥, [-0x18, -0x18], ⊥)
A19	`[ebp-0xc]`(⊥, [-0x10, -0x10], ⊥)	(⊥, ⊥, [0x0, 0x0])
	`eax`	(⊥, ⊥, [0x0, 0x0])
A20	`eax`	(⊥, ⊥, [0x4, 0x4])
A21	`[eax]`(⊥, ⊥, [0x4, 0x4])	([0x0, 0x0], ⊥, ⊥)

- Functions that only write memory in its own stack, TLS variables, and meta-data regions (*e.g.,* `malloc`).

For any memory access operated by these functions, we can, therefore, immediately determine the potential region(s).

4.1.2 *Value-set Analysis for single trace*

After memory regions are identified, we also customize the approach to assign the corresponding value sets to memory accesses and propagate them through execution trace. As execution trace specifies one path in the control flow graph (CFG), it can avoid the problem that all successors of indirect jumps and indirect calls cannot be identified in the original VSA implementation. Thus, we introduce how to do Value-set Analysis on a single trace.

VSA tracks down variable-like entities referred to as *a-locs*. By convention, an *a-loc* could be a register, a memory cell on the stack, on the heap, or in the global region. Particularly, VSA represents a memory *a-loc* as a combination of the value held by that memory cell and the value set indicating the address of that memory cell, as is shown in Table 1. For example, for `A5`: `mov [ebp-0xc], eax`, VSA specifies its corresponding *a-loc* as `[ebp-0xc]`(⊥, [-0x10, -0x10], ⊥). Here, `[ebp-0xc]` indicates the name of the stack memory cell, and (⊥, [-0x10, -0x10], ⊥) is the value set of the memory address.

For each *a-loc* identified, VSA computes a value set, indicating the set of values that each *a-loc* could potentially equal to. By convention, VSA represents such a value set as a *n-tuple* pertaining to *n* regions partitioned. For each element in the tuple, VSA specifies a range of offsets which indicates the values that the *a-loc* may equal to with respect to the corresponding region. To illustrate this, we take the register *a-loc* `esp` as an example. As depicted in Table 1 line 1, VSA specifies its value set as a *3-tuple* `(global ↦ ⊥, stack ↦ [-0x4, -0x4], heap ↦ ⊥)`, for brevity (⊥, [-0x4, -0x4], ⊥). In this set, ⊥ is a symbol – denoting the empty set of offsets (*i. e.,* ∅) – reflecting the fact that the register `esp` cannot refer to any memory cells on the heap or global region. Since instruction `push` offsets `esp` by 4 from the starting point of the stack, VSA assigns the value set [-0x4, -0x4] to the register *a-loc* `esp`, and attaches this set to the stack. Value sets are propagated with the following instruction trace in such an instinctive arithmetical way.

We assume that code in Figure 2 represents the complete execution trace of a program and draw Table 1 to indicate the value set tied to each of the *a-locs* identified from the assembly code. With this table, we then perform memory alias analysis by examining all of the value sets of addresses attached to memory *a-locs*. We can easily observe that memory `[eax]` at `A14` and `[eax]` at `A21` are the only pair of memory aliases pertaining to the execution trace. This is simply because the *a-locs* tied to these two memory segments are the only pair that carries the overlapping value set corresponding to their addresses, *i. e.,* (⊥, ⊥, [0x4, 0x4]).

In the above example, VSA exhibits perfect performance in alias analysis. However, this does not imply that VSA could perfectly resolve the memory alias issue in the context of postmortem program analysis. Generally, the execution trace logged for failure diagnosis has a limited length, indicating only a partial execution chronology prior to a program crash. The incomplete trace directly leads to the potentially imprecise data flow in VSA.

To address the above problem, we ensure that the instruction trace covers the function entry. Considering that there may exist multiple paths from the entry to the trace beginning and we have no idea of the exact path, we conservatively consider all possibilities - we propagate the value sets along all possible paths from the entry to the trace beginning. This guarantees a sound result. In the example shown in Figure 2, there is a single path from the entry to the trace beginning. Therefore, we can easily reconstruct the full path and obtain a better data flow. Note that we follow the original VSA to perform inter-procedure analysis if the extended instructions contain function calls.

4.2 Reverse Execution

Here, we describe the algorithm that `POMP++` follows when performing reverse execution. In particular, our algorithm follows two steps – *use-define chain construction* and *memory alias verification*. In the following, we elaborate on them in turn.

4.2.1 *Use-Define Chain Construction*

In the first step, the algorithm first parses an execution trace reversely. For each instruction in the trace, it extracts uses and definitions of corresponding variables based on the semantics of that instruction and then links them to a use-define chain previously constructed. For example, given an initial use-define chain derived from instructions `A21-A19`, `POMP++` extracts the uses and definitions from instruction `A18` and links them to the head of the chain (see Figure 3).

As is shown in the figure, a definition (or use) includes three elements – instruction ID, use (or definition) specification and the value of the variable. In addition, we can observe that a use-define relation includes not only the relations between operands but also those between operands and those base and index registers enclosed (see the use and definition for instruction `A19` shown in Figure 3).

Every time appending a use (or definition), our algorithm examines the reachability for the corresponding variable and attempts to resolve those variables on the chain. More specifically, it checks each use and definition on the chain and determines if the value of the corresponding variable can be resolved. By resolving,

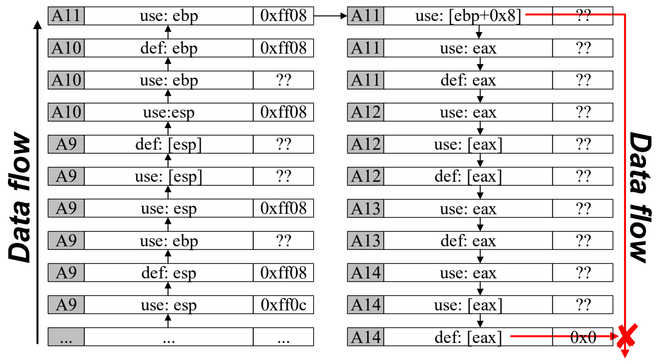


Fig. 4. A use-define chain with one intervening tag conservatively placed. The tag blocks the propagation of some data flows. Note that ✗ represents the block of a data flow.

we mean the variable satisfies one of the following conditions – (1) the definition (or use) of that variable can reach the end of the chain without any other intervening definitions; (2) it can reach its consecutive use in which the value of the corresponding variable is available; (3) a corresponding resolved definition at the front can reach the use of that variable; (4) the value of that variable can be directly derived from the semantics of that instruction (*e.g.,* variable `eax` is equal to `0x00` for instruction `mov eax, 0x00`).

To illustrate this, we take the example shown in Figure 3. After our algorithm concatenates definition `def:esp=esp+4` to the chain, where most variables have already been resolved, reachability examination indicates this definition can reach the end of the chain. Thus, the algorithm retrieves the value from the post-crash artifact and assigns it to `esp` (see the value in circle). After this assignment, our algorithm further propagates this updated definition through the chain, and attempts to use the update to resolve variables, the values of which have not yet been assigned. In this case, our algorithm further appends `use:esp` whose variable `esp` is not resolvable through the aforementioned reachability examination. Therefore, our algorithm derives the value of `esp` from the semantics of instruction `A18` (*i. e.,* `esp=esp-4`).

During use-define chain construction, our algorithm also keeps track of constraints in two ways. In one way, our algorithm extracts constraints by examining instruction semantics. In another way, our algorithm extracts constraints by examining use-define relations. In particular, (1) when the definition of a variable can reach its consecutive use without intervening definitions, our algorithm extracts a constraint indicating the variable in that definition shares the same value with the variable in the use. (2) When two consecutive uses of a variable encounter no definition in between, our algorithm extracts a constraint indicating variables in both uses carry the same value. (3) With a variable resolved, our algorithm extracts a constraint indicating that variable equals to the resolved value. The reason behind the maintenance of these constraints is to be able to perform memory alias verification discussed in the following section.

In the process of resolving variables and propagating definitions (or uses), our algorithm typically encounters a situation where an instruction attempts to assign a value to a variable represented by a memory cell but the address of that region cannot be resolved by using the information on the chain. For example, instruction `A14` shown in Figure 2 represents a memory write, the address of which is indicated by register `eax`. From the use-define chain pertaining to this example shown in Figure 4, we can easily observe the node with `A13 def:eax` does not carry any value though its impact can be propagated to the node with `A14 def:[eax]` without any other intervening definitions.

When this situation appears, a definition like `A14 def:[eax]` may potentially interrupt the reachability of the definitions and uses of other variables represented by memory accesses. For example, as is described in Section 3 that memory indicated by `[ebp+0x08]` and `[eax]` might be an alias of each other, definition `A14 def:[eax]` may block the reachability of `A11 use:[ebp+0x08]`. As such, in the step of use-define chain construction, our algorithm treats those unknown memory writes as an intervening tag and blocks previous definitions and uses accordingly. This conservative design strategy ensures that our algorithm does not introduce errors to memory footprint recovery. Note that once such intervening tags are resolved by our alias verification (which will be shortly explained), we will re-construct the blocked use-def chains.

In addition, during the execution of a program, it may invoke a system call, which traps execution into kernel space. As we will discuss later, we do not trace execution in the kernel space. In order to remedy the loss of such execution trace, we study the influences that system calls have on the user space and deal with them in a different manner. And a majority of system calls do not incur modification to registers and memory in user space and need no attention. If system calls may influence registers holding a value for a crashing program, our algorithm simply introduces definitions on the use-define chain. For system calls that can manipulate memory cells in the user space, our algorithm identifies the memory area influenced by that system call. Due to the calling convention of system calls on Linux, the affected memory area depends on the arguments of system calls or some specific registers. Thus, if our algorithm identifies the size of that memory region, it will append definitions to the chain accordingly. Otherwise, our algorithm treats that system call as an intervening tag which blocks the propagation.

The above forward-and-backward analysis is mainly designed to discover the use-define relations. Other techniques, such as static program slicing [20], can also identify use-define relations. However, our analysis is novel. To be specific, our analysis discovers the use-define relations and use them to perform the restoration of memory footprints. In turn, it leverages recovered memory footprints to further find use-define relations. This interleaving approach leads more use-define relations to being identified. More details about how we resolve memory alias are presented in the next section.

4.2.2 *Memory Alias Verification*

The second step of our algorithm is to minimize the side effect of memory aliases, *i. e.,* to examine if there exists any overlapping between the memory areas that a pair of symbolic names points to.

We have described the method of hypothesis testing in Section 3. During the process, the use-define chain is changed according to the hypothesis and the newly propagated data flow comes with two types of conflicts. The example of `[ebp+0x08]` in `A11` and `[eax]` in `A14` in Figure 2 shows the most common type, inconsistent data dependency. Another type is invalid data dependency in which a variable carries an invalid value that is supposed to make the crashing program terminate earlier or follow a different execution path. Once a constraint conflict is observed, our algorithm can easily reject the corresponding hypothesis and deem the pair of symbolic names is alias (or non-alias) of each other. Otherwise, if none of these hypotheses produces constraint conflicts, the proceeding of reverse execution will be impeded because we cannot reject any hypothesis.

In Figure 2, HT exposes this weakness when encountering the example of `[ebp+0x08]` in `A11` and `[eax]` in `A12` as we mentioned before. The unobserved conflicts like this imply that there is a lack of evidence against our hypothesis test, decreasing the mechanism's effectiveness. Meanwhile, HT seeks conflict recursively along the use-define chain, incurring exponential computation cost. As VSA is of benefit to providing extra static information and reducing performance overhead, we augment reverse execution with it. In our design, we explore three possible schemes to incorporate VSA:

VSA. The most intuitive reaction is to replace HT with VSA. Given the instruction trace pertaining to a crash, we do VSA to compute the address set of each memory access. During the time of reverse execution, we verify the alias relation between two memory cells by comparing their address sets. Zero overlap means non-alias, and two identical single-value sets indicate alias. For all the remaining cases, we regard their alias relation as unknown. Despite this scheme has low complexity, as we will demonstrate in our evaluation, it is limited in that its static nature will lead to less data flow information and ultimately reduce the effectiveness of diagnosis.

VSA+HT. In this scheme, we combine VSA and HT for alias verification. Following the proceeding of reverse execution, when encountering an uncertain alias, we first query VSA in the manner as explained above. If the alias relation remains unknown, we then switch to HT. The intuition behind this scheme is that (1) we can combine the strengthens of both VSA and HT to achieve the best effectiveness and (2) using VSA earlier will save a large volume of HT to accelerate the verification process. Not surprisingly, our evaluation shows that this scheme enhances not only the effectiveness but also the efficiency.

Fixed Point. By intuition, the information recovered in reverse execution may refine the results of VSA, which probably in turn improve alias verification. Following this idea, we explored another scheme (Algorithm 1) to incorporate VSA. Given an instruction trace, we start with a round of reverse execution without alias verification, building a preliminary data flow. Then we perform a recursive analysis. In each iteration, we re-calculate the VSA results with available run-time information and then do a round of reverse execution following `VSA+HT`. We end this analysis when an iteration cannot update the data flow (*i. e.,* reaching a fixed point). In theory, this scheme fully releases the potential of VSA and HT, and indeed maximizes the recovery of data flow. Nevertheless, we observe that this scheme shows the same level of diagnosis utility as `VSA+HT`, while incurring substantially more computation cost.

In summary, the three schemes can all systematically incorporate VSA. However, only `VSA+HT` can achieve both improvement from the perspectives of data flow recovery and efficiency. Therefore, by default, we enable `POMP++` to work with the `VSA+HT` scheme. Note that `POMP++` can be configured to run `VSA` and `Fixed Point` in case of need.

4.3 Backward Taint Analysis

Recall that the goal of this work is to pinpoint instructions truly pertaining to a program crash. In Section 3, we briefly introduce how backward taint analysis plays a role in achieving this goal. Here, we describe more details.

To perform backward taint analysis, `POMP++` first identifies a sink. In general, a program crash results from two situations – executing an invalid instruction or dereferencing an invalid address. For the first situation, `POMP++` deems the program counter (`eip`) as a sink because executing an invalid instruction indicates `eip` carries a bad value. For the second situation, `POMP++` treats a general register as a sink because it holds a value which points to an invalid address, like `[eax]` in Figure 2.

With a sink identified, `POMP++` taints the sink and performs taint propagation backward. `POMP++` looks up the aforementioned use-define chain to identify the definitions of the tainted variables and selects them following the rule of reaching definition without intervention. Not only the accessed memory of the tainted variables but also base and index registers (if available) should be tainted. So, with sink `[eax]` serving as the initial tainted variable, `POMP++` selects `A14 def:[eax]=0x0` on the chain as they are a pair of memory aliases. In addition, `A20 def:eax=eax+0x4`, `A19 def:eax=[ebp-0xc]`, *etc.* , are tainted successively according to the propagation strategy. By doing so, `POMP++` is guaranteed never to miss the root cause of a program crash though it over-taints some variables that do not actually contribute to the crash.

Similar to the situation seen in reverse execution, when performing taint propagation backward, `POMP++` may encounter a definition on the chain which intervenes the propagation. For example, given a tainted variable `[`R_0`]` and a definition `def:[`R_1`]` with R_1 unknown, `POMP++` cannot determine whether R_0 and R_1 share the same value and `POMP++` should pass the taint to variable `[`R_1`]`. When this situation appears, `POMP++` follows the idea of the aforementioned approach and examines if both variables share the same address. When "fail-to-reject" occurs, therefore, `POMP++` over-taints the variable in that intervening definition. Again, this can ensure that `POMP++` does not miss the enclosure of the root cause.

Algorithm 1 Recursive VSA-enhanced reverse execution algorithm, namely `Fixed Point`. Each round of reverse execution only performs one layer of `VSA+HT`.

```
Input:
    The instruction trace, $inst\_trace$;
    The core dump, $core\_dump$;
Output:
    Data flow along the instruction trace, $data\_flow$;
1: $data\_flow = 0$
2: $temp\_df = 0$
3: $temp\_df = rev\_exe\_noaliasverify(temp\_df)$
4: while $temp\_df > data\_flow$ do
5:     $data\_flow = temp\_df$
6:     $temp\_df = calculate\_vsa(temp\_df)$
7:     $temp\_df = do\_vsa\_ht(temp\_df)$
8: end while
9: $data\_flow = temp\_df$
```

5 Implementation

We have implemented a prototype of `POMP++` for Linux 32-bit system with Linux kernel 4.4 running on an Intel i7-6700HQ quad-core processor (a 6th-generation Skylake processor) with 16 GB RAM. Our prototype consists of three major components – ① a sub-system that implements Value-set Analysis on single trace, ② a sub-system that implements the aforementioned reverse execution and backward taint analysis, and ③ a sub-system that traces program execution with Intel PT. In total, our implementation carries about 30,000 lines of `C` code. In the following, we present some important implementation details.

Following the design description above, we implemented `84` distinct instruction handlers to perform reverse execution, backward tainting. Furthermore, we add new instruction handlers to calculate value set of all the `a-locs`. Along with these handlers, we also built core dump and instruction parsers on the basis of `libelf` [21] and `libdisasm` [22], respectively. Note that for instructions with the same semantics (*e.g.,* je, jne, and jg) we dealt with their inverse operations in one unique handler. To keep track of constraints and perform verification, we reuse the Z3 theorem prover [23, 24].

To allow Intel PT to log execution in a *correct* and *reliable* manner, we implemented the second sub-system as follows. We enabled Intel PT to run in the Table of Physical Addresses (ToPA) mode, which allows us to store PT packets in multiple discontinuous physical memory areas. We added to the ToPA an entry that points to a `16 MB` physical memory buffer. In our implementation, we use this buffer to store packets. To be able to track if the buffer is fully occupied, we clear the `END` bit and set the `INT` bit. With this setup, Intel PT can signal a performance-monitoring interrupt at the moment the buffer is fully occupied. Moreover, we intercepted the context switch to enable `POMP++` to examine the threads in/out, and store PT packets for each thread.

Considering the Intel CPU utilizes Supervisor Mode Access Prevention (SMAP) to restrict the access from kernel to user space, our implementation toggles SMAP between packet migration. In addition, we configured Intel PT to exclude packets irrelevant to control flow switching (*e.g.,* timing information) and paused its tracing when execution traps into kernel space. In this way, `POMP++` is able to log an execution trace sufficiently long. Last but not least, we introduced new resource limit `PT_LIMIT` into the Linux kernel. With this, software developers and security analysts can not only select which process to trace, but also configure the size of the circular buffer in a convenient manner.

Recall that our customized VSA often needs to extend the truncated trace beginning to cover the full function. This extending operation requires analysis over the control flow graph (CFG) of the first function. To support that, we reuse the CFG-recovery utility shipped with the open source project `angr` [25].

6 EVALUATION

In this section, we demonstrate the utility of `POMP++` using the crashes resulting from real-world vulnerabilities. To be more specific, we present the effectiveness and efficiency of `POMP++`, and discuss those crashes that `POMP++` fails to handle properly.

6.1 Experiment Setting

6.1.1 Experiment Setup

To demonstrate the utility of `POMP++`, we selected 30 programs and benchmarked `POMP++` with their crashes resulting from 33 real-world PoCs obtained from Offensive Security Exploit Database Archive [26]. Table 2 shows these crashing programs and summarizes the corresponding vulnerabilities. As we can observe, the selected programs cover a wide spectrum ranging from sophisticated software like `BinUtils` with lines of code over `690K` to lightweight software such as `stftp` and `psutils` with lines of code less than `2K`.

Regarding vulnerabilities resulting in the crashes, our test corpus encloses not only 28 memory corruption vulnerabilities (stack/heap overflow, integer overflow, use-after-free) but also 5 common software defects like null pointer dereference and invalid free. The reason behind this selection is to demonstrate that, beyond memory corruption vulnerabilities, `POMP++` can be generally applicable to other kinds of software defects.

6.1.2 Experimental Design

For each program crash shown in Table 2, we performed manual analysis with the goal of finding out the minimum set of instructions that truly contribute to that program crash. We took our manual analysis as ground truth and compared them with the output of `POMP++`. In this way, we validated the effectiveness of `POMP++` in facilitating failure diagnosis. More specifically, we compared the instructions identified manually with those pinpointed by `POMP++`. The focuses of this comparison include (1) examining whether the root cause of that crash is enclosed in the instruction set `POMP++` automatically identified, (2) investigating whether the output of `POMP++` covers the minimum instruction set that we manually tracked down, and (3) exploring how many memory addresses could be resolved in each scheme.

In order to evaluate the efficiency of `POMP++`, we recorded the time of different schemes they took when spotting the instructions that truly pertain to each program crash for further comparison.

Considering pinpointing a root cause does not require reversely executing the entire trace recorded by Intel PT, it is noteworthy that, we selected and utilized only a partial execution trace for evaluation. In this work, our selection strategy follows an iterative procedure in which we first introduced instructions of a crashing function to reverse execution. If this partial trace is insufficient for spotting a root cause, we traced back functions previously invoked and then included instructions function-by-function until that root cause can be covered by `POMP++`.

6.2 Experimental Results

6.2.1 Effectiveness

As a postmortem program analysis, the most intuitive utility indicator of `POMP++` is its effectiveness in tracking down the root cause. To serve this evaluation, we perform four sets of experiments, one performing reverse execution with HT only and others performing reverse execution with the three schemes described in Section 4 respectively. As shown in Table 2, the two hybrid schemes, `VSA+HT` and `Fixed Point`, have better utilities than `HT` or `VSA` alone in identification of root causes. Other than the cases that are missed by all the schemes (`aireplay-ng-1.2b3` and `0verkill-0.16`), our hybird schemes capture the root cause in all the cases. However, `HT` additionally misses the cases of `gdb-7.5.1` and `JPegToAvi-1.5` and `VSA` additionally misses the cases of `unrar-3.9.3` and `JPegToAvi-1.5`. In the following we briefly explain those cases.

gdb-7.5.1. Figure 5 shows the propagation of corrupted memory that leads to the crash in `gdb-7.5.1`. Specifically, a defect at instruction `1` corrupts the memory indexed by `[edi]`. The bad memory propagates from instruction `1` to instruction `6` (`[edi]` at instruction `1` and `[esp+0xC]` at instruction `6` are alias) and then through instruction `6` (`[esp+0xC]` to `ebx`), instruction `7` (`ebx` to `[esp]`), instruction `9` (`[esp]` to `esi`), until instruction `10` (bad value in `esi` is used as memory address). `HT` fails to capture the propagation from instruction `1` to instruction `6`. This is because the memory write to `[eax]` at instruction `3` has an unknown address, and `HT` cannot determine the alias relation

TABLE 2
The list of program crashes resulting from various vulnerabilities. `CVE-ID` specifies the ID of the CVEs. `Trace Length` indicates the lines of instructions that `POMP++` reversely executed. `HT` indicates reverse execution with HT only, and `VSA`, `VSA+HT`, and `Fixed Point` are the three schemes of VSA-enhanced reverse execution described in Section 4.2.2. `Mem addr known(%)` illustrates the percentage of memory locations, the addresses of which are resolvable. `Mem addr inc(%)` with **+/-** means the corresponding percentage differences compared with `Mem addr known(%)` of `HT`.

Vulnerability			***Diagnose Results***								
Name	LoC	CVE-ID	Trace Length	HT		VSA		VSA + HT		Fixed Point	
				Root Cause	Mem addr known (%)	Root Cause	Mem addr inc (%)	Root Cause	Mem addr inc (%)	Root Cause	Mem addr inc (%)
coreutils-8.4	138135	2013-0223	4000	✓	85.89%	✓	+0.00%	✓	+6.70%	✓	+10.80%
coreutils-8.4	138135	2013-0222	5867	✓	38.76%	✓	-0.22%	✓	+9.55%	✓	+14.90%
coreutils-8.4	138135	2013-0221	5015	✓	99.44%	✓	+0.00%	✓	+0.00%	✓	+0.08%
mcrypt-2.5.8	37439	2012-4409	1350	✓	87.45%	✓	-19.37%	✓	+0.00%	✓	+1.11%
BinUtils-2.15	697354	2006-2362	867	✓	100.00%	✓	-1.57%	✓	+0.00%	✓	+0.00%
unrtf-0.19.3	5039	NA	1085	✓	100.00%	✓	+0.00%	✓	+0.00%	✓	+0.00%
psutils-p17	1736	NA	3123	✓	62.57%	✓	-1.90%	✓	+6.16%	✓	+13.63%
stftp-1.1.0	1559	NA	3651	✓	98.77%	✓	-3.03%	✓	+0.79%	✓	+0.79%
nasm-0.98.38	33553	2004-1287	4064	✓	30.73%	✓	+62.40%	✓	+69.27%	✓	+69.27%
libpng-1.2.5	33681	2004-0597	4214	✓	56.63%	✓	+1.30%	✓	+8.69%	✓	+13.30%
putty-0.66	90165	2016-2563	7338	✓	72.32%	✓	-4.38%	✓	+1.03%	✓	+2.78%
Unalz-0.52	8546	2005-3862	61999	✓	54.28%	✓	+0.01%	✓	+20.96%	✓	+36.03%
LaTeX2rtf-1.9	14473	2004-2167	17056	✓	30.59%	✓	+49.76%	✓	+49.76%	✓	+49.76%
aireplay-ng-1.2b3	62656	2014-8322	18569	✗	72.81%	✗	+0.40%	✗	+0.40%	✗	+0.04%
corehttp-0.5.3a	914	2007-4060	25385	✓	100.00%	✓	+0.00%	✓	+0.00%	✓	+0.00%
gas-2.12	595504	2005-4807	25713	✓	34.89%	✓	+0.51%	✓	+20.61%	✓	+35.39%
abc2mtex-1.6.1	4052	NA	29521	✓	56.79%	✓	+0.25%	✓	+17.77%	✓	+28.28%
LibSMI-0.4.8	80461	2010-2891	50787	✓	86.43%	✓	-4.70%	✓	+0.04%	✓	+0.07%
gif2png-2.5.2	1331	2009-5018	70854	✓	43.31%	✓	-0.01%	✓	+8.02%	✓	+16.40%
O3read-0.03	932	2004-1288	78244	✓	54.42%	✓	-0.08%	✓	+13.80%	✓	+19.75%
unrar-3.9.3	17575	NA	36216	✓	95.29%	✗	-4.34%	✓	+0.00%	✓	+4.71%
nullhttp-0.5.0	1849	2002-1496	460	✓	94.38%	✓	+0.00%	✓	+0.00%	✓	+0.00%
nginx-1.4.0	100255	2013-2028	158	✓	100.00%	✓	+0.00%	✓	+0.00%	✓	+0.00%
Python-2.2	416060	2007-4965	3426	✓	87.44%	✓	+0.00%	✓	+1.20%	✓	+2.83%
0verkill-0.16	16361	2006-2971	10494	✗	nan	✗	nan	✗	nan	✗	nan
openjpeg-2.1.1	169538	2016-7445	1035	✓	41.30%	✓	+0.00%	✓	+1.19%	✓	+18.50%
podofo-0.9.4	60147	2017-5854	42165	✓	99.88%	✓	-2.14%	✓	+0.12%	✓	+0.12%
Python-2.7	906829	NA	551	✓	100.00%	✓	+0.00%	✓	+0.00%	✓	+0.00%
poppler-0.8.4	183535	2008-2950	672	✓	100.00%	✓	+0.00%	✓	+0.00%	✓	+0.00%
Ntpd-4.2.6	152433	NA	1766	✓	52.42%	✓	-1.10%	✓	+0.74%	✓	+2.20%
prozilla-1.3.6	13070	2004-1120	5560	✓	34.85%	✓	+1.79%	✓	+1.92%	✓	+2.25%
gdb-7.5.1	1651764	NA	4009	✗	69.69%	✓	+6.97%	✓	+9.5%	✓	+25.05%
JPegToAvi-1.5	580	NA	133734	✗	46.91%	✗	-6.22%	✓	+12.56%	✓	+15.01%

```
 1  stos es:[edi], eax ; root cause
 2  ......
 3  mov [eax], ecx ; unknown memory access
 4  xor eax, eax
 5  ......
 6  mov ebx, [esp+0xC]
 7  mov [esp], ebx
 8  ......
 9  mov esi, [esp]
10  mov ecx, [esi] ; crashing instruction
```

Fig. 5. Propagation of corrupted memory leading to the crash in `gdb-7.5.1`

between that memory write and `[esp+0xC]` at instruction `6` using the recovered dynamic information. Hence, `HT` assumes `[edi]` at instruction `1` may get overwritten at instruction `3`, which would not flow to instruction `6`. By contrast, `VSA` has no such difficulty as `[esp+0xC]` at instruction `6` accesses the stack and `[eax]` accesses the heap (the index is propagated from the return value of `malloc`).

unrar-3.9.3. We present the corrupted memory propagation of this case in Figure 6. The memory cell `[eax]` at instruction `1` is corrupted. It then propagates through the following path: from instruction `1` to instruction `8` (`[eax]` and `[esp+0x4]` are aliases), from instruction `8` to instruction `9` (flow carried by `eax`), from instruction `9` to instruction `11` (`[ecx]` and `[ebx]` are aliases), from instruction `11` to instruction `12` (flow carried by `esi`). Using pure static analysis, `VSA` infers that `edx` after

```
 1  mov byte ptr [eax], al ; root cause
 2  ......
 3  and edx, edi ; edx:meaningless value set
 4  mov [edx], ebx ; unknown memory access
 5  mov [edx+0x4], eax ; unknown memory access
 6  ......
 7  ......
 8  mov eax, [esp+0x4]
 9  mov [ecx], eax
10  ......
11  mov esi, [ebx]
12  mov eax, [esi] ; crashing instruction
```

Fig. 6. Propagation of corrupted memory leading to the crash in `unrar-3.9.3`

instruction `3` could carry arbitrary value, failing to understand the addresses of the two memory writes at instruction `4` and instruction `5`. Therefore, `VSA` cannot determine the propagation from `[eax]` at instruction `1` to `[esp+0x4]` at instruction `8`. Alternatively, `HT` can learn the concrete value of `edx` after instruction `3` from the recovered information and hence, avoid the problem encountered by `VSA`.

JPegToAvi-1.5. In this case, the memory corrupted at instruction `1` (*i. e.*, `[ebx]`) propagates to the crash site through instruction `10`, instruction `12`, instruction `14` and instruction `15` (as shown in Figure 7). To determine the first hop in the propagation, we need to know that `[ebx]` at instruction `1` and `[esp]` at instruction `10` are aliases and the two memory accesses neither alias with `[ecx]` at instruction `3` nor `[eax]` at instruction `7`. However, `VSA` can only

```
 1  mov byte ptr [ebx], cl ; root cause
 2  ......
 3  mov [ecx], ebx ; unknown memory access
 4  xor ecx, ecx
 5  ......
 6  and eax, [ebx] ; eax:meaningless value set
 7  mov [eax], edi ; unknown memory access
 8  mov [esp+0x1C], eax
 9  ......
10  mov esi, [esp]
11  lea ebx, [esp+0x14]
12  mov [ebx], esi
13  ......
14  mov eax, [esp]
15  mov eax, [eax] ; crashing instruction
```

Fig. 7. Propagation of corrupted memory leading to the crash in `jpegtoavi-1.5`

infer the non-alias relation between `[esp]` at instruction `10` and `[ecx]` at instruction `3` (`[esp]` accesses the stack while `[ecx]` accesses the heap), and `HT` can only determine the other required non-alias relation. Therefore, `VSA` or `HT` alone fails in this case, but combining them two can succeed.

Going beyond the identification of root causes, a more general metric of reverse execution is data flow recovery [1, 10]. The reason is that identification of root causes actually depends on small pieces of data flow, carrying high randomness as an evaluation metric. To evaluate `POMP++` in a more objective manner, we perform another measurement, in which we quantify the data flow rebuilt by each method. To be more specific, when the enhanced reverse execution terminates, we count the percentage of memory accesses with known addresses. We believe this percentage well represents the comprehensiveness of data flow building, because it essentially corresponds to the extension that we understand use-define relations. The measurement results are shown in Table 2. From Table 2, we can first observe that `VSA` is less effective in recovering memory footprints than `HT`. To be specific, there are `24` out of 33 cases in which `HT` surpasses or equals `VSA`. This fact indicates that pure `VSA` will possibly lead to more failures in root cause diagnosis and is not effective enough compared with `HT`. Meanwhile, the hybrid schemes `VSA+HT` and `Fixed Point`, which combine VSA and HT for alias verification, outperform the `HT` in all cases, respectively increasing the percentage of known memory address by 12% in 22 cases and by 16% in 25 cases. The reason behind the increase is that VSA solves more addresses that HT cannot handle, and so do HT.

Summarizing the above measurement, merely using VSA for reverse execution leads to reduced diagnosis capability and weak memory footprint recovery than using HT alone. Hybrid methods integrating VSA and HT achieve the best results of memory recovery and root cause diagnosis.

6.2.2 *Efficiency*

Going beyond the effectiveness improvement of reverse execution, we also aim to achieve better efficiency via introducing VSA. To better demonstrate the efficiency of `POMP++`, we measure the computation complexity of the above experiments and show the results in Table 3.

Although pure `VSA` takes significantly less time than the other three schemes in all cases shown in Table 3, its defects of lower recovery ability and root causes missing determine that it is not

TABLE 3
Measurement of reverse execution efficiency. The execution time of `HT` is listed as a baseline. The performance improvement of $\times_{\text{VSA}}$ denotes the ratio of execution time of `HT` to `VSA`. It is the same with $\times_{\text{VSA+HT}}$ and $\times_{\text{FixedPoint}}$.

Vulnerability		***Time***	***Performance Improvement***		
Name	CVE-ID	HT	$\times_{\text{VSA}}$	$\times_{\text{VSA+HT}}$	$\times_{\text{FixedPoint}}$
coreutils-8.4	2013-0223	4.9s	1.44	1.09	0.82
coreutils-8.4	2013-0222	36.6s	21.53	1.22	0.68
coreutils-8.4	2013-0221	8.6s	3.31	2.26	2.05
mcrypt-2.5.8	2012-4409	3s	4.29	1.30	0.79
BinUtils-2.15	2006-2362	3s	7.50	1.25	0.75
unrtf-0.19.3	NA	1m10s	14.00	1.27	0.88
psutils-p17	NA	4m	11.88	1.59	1.04
stftp-1.1.0	NA	4m	6.86	1.31	0.86
nasm-0.98.38	2004-1287	1m4s	16.00	10.67	9.14
libpng-1.2.5	2004-0597	5m46s	1.97	1.09	0.96
putty-0.66	2016-2563	30m	163.64	15.00	6.00
Unalz-0.52	2005-3862	6h	59.02	4.29	1.16
LaTeX2rtf-1.9	2004-2167	8m4s	1.03	1.01	0.98
aireplay-ng-1.2b3	2014-8322	10m32s	1.89	1.30	1.27
corehttp-0.5.3a	2007-4060	52m	32.16	4.33	2.60
gas-2.12	2005-4807	46m7s	4.24	4.02	3.51
abc2mtex-1.6.1	NA	2h	30.64	5.31	3.16
LibSMI-0.4.8	2010-2891	5h5m	25.42	1.05	0.73
gif2png-2.5.2	2009-5018	31m	3.44	0.65	0.49
O3read-0.03	2004-1288	23m	4.58	1.10	0.72
unrar-3.9.3	NA	5m30s	5.32	1.81	1.43
nullhttp-0.5.0	2002-1496	0.2s	2.00	1.00	0.67
nginx-1.4.0	2013-2028	3.2s	1.33	0.97	0.94
Python-2.2	2007-4965	3m	2.17	1.36	0.90
0verkill-0.16	2006-2971	1s	1.00	1.00	1.00
openjpeg-2.1.1	2016-7445	0.3s	3.00	1.50	1.00
podofo-0.9.4	2017-5854	2m	2.14	1.09	0.82
Python-2.7	NA	0.2s	2.00	1.00	0.67
poppler-0.8.4	2008-2950	13s	2.60	1.63	1.18
Ntpd-4.2.6	NA	3.8s	12.67	0.76	0.54
prozilla-1.3.6	2004-1120	2m14s	1.60	1.03	0.93
gdb-7.5.1	NA	1m2s	1.11	1.05	0.78
JPegToAvi-1.5	NA	8h25m	8.29	6.10	1.37

suitable for the postmortem program analysis which focuses on pinpointing instructions pertaining to the crash. However, the fact that the execution time of `VSA` is `15x` less than `HT` still well supports our intuition that static analysis has superior efficiency.

Thus, we focus on the measurement of the time complexity of two hybrid methods. On average, `VSA+HT` reduces 60% of execution time compared with the single `HT`, while `Fixed Point` executes even more time than `HT`. By intuition, they shall avoid a great number of hypothesis testing and would reduce the computation complexity. In 30 cases of `VSA+HT` and 14 cases of `Fixed Point`, our evaluation shows that the combination takes less or equal time than using hypothesis testing only. For cases in which hypothesis testing takes less time, we believe it is mainly because VSA recovers more information that can be consumed by hypothesis testing and in turn triggers further space of reverse execution. This is actually supported by the above effectiveness evaluation.

To sum up, though `Fixed Point` recovers the most memory footprints, it involves the most computations complexity which is unacceptable. The other hybrid method `VSA+HT` can achieve a great balance between effectiveness and efficiency, *i. e.,* maintain similarly high recovery ability to `Fixed Point` and reduce execution time massively. Thus, `POMP++` chooses to implement `VSA+HT` for augment by default. If needed, we additionally provide configurations of `VSA` for time-critical scenarios and `Fixed Point` for accuracy-critical ones.

7 Related Work

This research work mainly focuses on locating software vulnerability from its crash dump. Regarding the techniques we employed and the problems we addressed, the lines of works most closely related to our own include reverse execution and postmortem program analysis, static analysis for binary-level memory alias. In this section, we summarize previous studies and discuss their limitation in turn.

Reverse execution. Reverse execution is a conventional debugging technique that allows developers to restore the execution state of a program to a previous point. Pioneering research [27–30] in this area relies upon restoring a previous program state from a record, and thus their focus is to minimize the amount of records that one has to save and maintain in order to return a program to a previous state in its execution history. In addition to state saving, program instrumentation is broadly used to facilitate the reverse execution of a program. For example, Hou *et al.* designed compiler framework `Backstroke` [31] to instrument C++ program in a way that it can store program states for reverse execution.

Given that state saving requires extra memory space and program instrumentation results in a slower forward execution, recent research proposes to employ a core dump to facilitate reverse execution. In [2] and [32], new reverse execution mechanisms are designed in which the techniques proposed reversely analyze code and then utilize the information in a core dump to reconstruct the states of a program prior to its crash. Since the effectiveness of these techniques highly relies upon the integrity of a core dump, and exploiting vulnerabilities like buffer overflow and dangling pointers corrupt memory information, they may fail to perform reverse execution correctly when memory corruption occurs.

Different from the prior research works discussed above, the reverse execution technique introduced in this paper follows a completely different design principle, and thus it provides many advantages. First, it can reinstate a previous program state without restoring that state from a record. Second, it does not require any instrumentation to a program, making it more generally applicable. Third, it is effective in performing execution backward even though the crashing memory snapshot carries corrupted data.

Postmortem program analysis. Liblit *et al.* proposed a backward analysis technique for crash analysis [33]. To be more specific, they introduced an efficient algorithm that takes as input a crash point as well as a static control flow graph, and computes all the possible execution paths that lead to the crash point. As is mentioned earlier, memory information may be corrupted when attackers exploit a program. The technique described in [33] highly relies upon the integrity of the information resided in memory, and thus fails to analyze program crash resulting from malicious memory corruption. In this work, we directly identify the root cause of software failures by reversely executing the program and reconstructing memory footprints prior to the crash.

Considering the low cost of capturing core dumps, prior studies also proposed to use core dumps to analyze the root causes of software failures. Of all the works along this line, the most typical ones include `CrashLocator` [34], `!analyze` [3] and `RETracer` [2] which locate software defects by analyzing memory information resided in a core dump. As such, these techniques are not suitable to analyze crashes resulting from malicious memory corruption. Different from these techniques, Kasikci *et al.* introduced `Gist` [35], an automated debugging technique that utilizes off-the-shelf hardware to enhance core dump and then employs a cooperative debugging technique to perform root cause diagnosis. While `Gist` demonstrates its effectiveness on locating bugs from a software crash, it requires the collection of crashes from multiple parties running the same software and suffering the same bugs. This could significantly limit its adoption. In our work, we introduce a different technical approach which can perform analysis at the binary level without the participation of other parties.

In recent research, Xu *et al.* [7] introduced `CREDAL`, an automatic tool that employs the source code of a crashing program to enhance core dump analysis and turns a core dump to an informative aid in tracking down memory corruption vulnerabilities. While sharing a common goal with our system – pinpointing the code statements where a software defect is likely to reside – `CREDAL` follows a completely different technical approach. More specifically, `CREDAL` discovers the mismatch in variable values and deems the code fragments corresponding to the mismatch as the possible vulnerabilities that lead to the crash. In this work, `POMP++` precisely pinpoints the root cause of vulnerability by utilizing the memory footprints recovered from reverse execution.

REPT introduced by Cui *et al.* [10] reconstructed the execution states with high fidelity by combining online Intel Processor Tracing with offline binary analysis which recovers previous data flow. To be specific, it performs iterative backward and forward binary analysis to detect and correct the inconsistency. And REPT could handle concurrent programs by timing information obtained from Intel PT. Instead of error correction, `POMP++` leverages Value-set Analysis to facilitate reverse execution with more accurate and precise memory alias result. With iteratively combining reverse execution and static Value-set Analysis, in theory, `POMP++` could recover more data flow than REPT.

Static analysis for binary-level memory alias. There is a long history of research about analyzing memory alias in binary code. As pioneering research works, Debray *et al.* [16] and Cifuentes *et al.* [36] both proposed the same type of technical approaches that compute the values a set of registers can hold at each program point and then use the values held in the registers to determine alias. Considering such techniques determine only the possible values held in each register, but do not reason about values across memory operations, Brumley *et al.* proposed a logic-based approach which derives all possible alias relationships by finding an over-approximation of the set of values that each memory location and register can hold at each program point [18]. At a high level, this logic-based approach is similar to Value-set Analysis [11, 12, 37] because they both perform value reasoning across memory operations. However, different from the work proposed in [18], Value-set Analysis neither assumes that all memory cells and register locations must be of a single fixed width, nor assumes reads and writes have to be no overlapping. As such, the value set analysis is more practical for real-world applications, whereas the logic-based approach [18] has been tested only against simple toy examples. So in this work, we propose `POMP++` which leverages VSA to analyze binary-level memory alias to facilitate postmortem program analysis.

8 Discussion

In this section, we discuss the limitations of our current design, insights we learned and possible future directions.

Multiple threads. `POMP++` focuses only on analyzing the post-crash artifact produced by a crashing thread, so we may miss vulnerabilities caused by thread interleaving. However, this does not mean the failure of `POMP++`, nor does it significantly downgrades the utility of `POMP++` due to following reasons. First, a prior study [38] has already indicated that a large fraction of software crashes involves only the crashing thread. Second, with integration of Intel PT timing information, we think POMP++ can synthesize a *complete* execution trace, making POMP++ work properly. As part of the future work, we will integrate this extension into the next version of POMP++.

Just-in-Time native code. POMP++ retrieves instructions from executable and library files by recorded program flow information. However, it fails to analyze binary code generated on the fly by Just-in-Time (JIT) mechanism. To make POMP++ handle programs in this type, in the future, we will augment POMP++ with the capability of tracing and logging native code generated at the run time. For example, we may monitor the executable memory and dump JIT native code accordingly.

9 Conclusion

In this paper, we develop POMP++ on Linux system to analyze post-crash artifacts. Especially, we make an effort to improve the ability to solve memory alias problem during reverse execution by introducing static analysis (*i. e.,* Value-set Analysis). We show that POMP++ can significantly reduce the manual efforts on the diagnosis of program failures, making software debugging more informative and efficient. Since the design of POMP++ is entirely on the basis of the information resided in a post-crash artifact, the technique proposed can be generally applied to diagnose the crashes of programs written in various programming languages caused by various software defects.

We demonstrated the effectiveness and efficiency of POMP++ using the real-world program crashes pertaining to 31 software vulnerabilities. We showed that POMP++ could reversely reconstruct the memory footprints of a crashing program and accurately and quickly identify the program statements (*i. e.,* instructions) that truly contribute to the crash. While POMP++ increases 12% of resolvable memory addresses, it reduces 60% of execution time, compared with the previous POMP. Following this finding, we safely conclude POMP++ can significantly downsize the program statements that a software developer (or security analyst) needs to manually examine.

Acknowledgment

We thank the anonymous reviewers for their helpful feedback on this paper. This work was supported by ARO W911NF-13-1-0421 (MURI), ARO W911NF-15-1-0576, NSF Award-1718459, NSF CNS-1505664, NSF CNS-1814679, and Chinese National Natural Science Foundation 61272078. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

[1] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, “Postmortem program analysis with hardware-enhanced post-crash artifacts,” in *26th USENIX Security Symposium (USENIX Security 17)*, USENIX Association, 2017.

[2] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, “Retracer: Triaging crashes by reverse execution from partial memory dumps,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[3] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, “Debugging in the (very) large: Ten years of implementation and experience,” in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[4] Y. Cao, H. Zhang, and S. Ding, “Symcrash: Selective recording for reproducing crashes,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[5] J. Bell, N. Sarda, and G. Kaiser, “Chronicler: Lightweight recording to reproduce field failures,” in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[6] S. Artzi, S. Kim, and M. D. Ernst, “Recrash: Making software failures reproducible by preserving object states,” in *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, 2008.

[7] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[8] P. Ohmann, “Making your crashes work for you (doctoral symposium),” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015.

[9] “Processor tracing.” https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.

[10] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, “REPT: Reverse debugging of failures in deployed software,” in *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, USENIX Association, 2018.

[11] G. Balakrishnan and T. W. Reps, “Analyzing memory accesses in x86 executables,” in *Proceedings of the 13th International Conference on Compiler Construction (CC)*, 2004.

[12] G. Balakrishnan and T. Reps, “Wysinwyx: What you see is not what you execute,” *ACM Transactions on Programming Languages and Systems*, 2010.

[13] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, “Rx: treating bugs as allergies—a safe method to survive software failures,” in *ACM SIGOPS Operating Systems Review*, vol. 39, pp. 235–248, ACM, 2005.

[14] W. Gu, Z. Kalbarczyk, R. K. Iyer, Z.-Y. Yang, *et al.*, “Characterization of linux kernel behavior under errors.,” in *DSN*, vol. 3, pp. 22–25, 2003.

[15] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, “Conseq: detecting concurrency bugs through sequential errors,” in *ACM SIGPLAN Notices*, vol. 46, pp. 251–264, ACM, 2011.

[16] S. Debray, R. Muth, and M. Weippert, “Alias analysis of executable code,” in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1998.

[17] M. Fernandez and R. Espasa, “Speculative alias analysis for executable code,” in *International Conference on Parallel Architectures and Compilation Techniques, 2002. Proceedings*, 2002.

[18] D. Brumley and J. Newsome, “Alias analysis for assembly,” in *CMU-CS-06-180*, 2006.

[19] O. Corporation, “Position-independent code.” https://docs.oracle.com/cd/E26505_01/html/E26506/glmqp.html.

[20] M. Weiser, “Program slicing,” in *Proceedings of the 5th international conference on Software engineering*, pp. 439–

449, IEEE Press, 1981.

[21] "Libelf - free software directory." https://directory.fsf.org/wiki/Libelf.

[22] "libdisasm: x86 disassembler library." http://bastard.sourceforge.net/libdisasm.html.

[23] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[24] "The z3 theorem prover." https://github.com/Z3Prover/z3.

[25] S. Yan, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, and C. Kruegel, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *Security and Privacy*, pp. 138–157, 2016.

[26] "Offensive security exploit database archive." https://www.exploit-db.com/.

[27] B. Biswas and R. Mall, "Reverse execution of programs," *SIGPLAN Not.*, 1999.

[28] T. Akgul and V. J. Mooney III, "Assembly instruction level reverse execution for debugging," *ACM Trans. Softw. Eng. Methodol.*, 2004.

[29] T. Akgul and V. J. Mooney, III, "Instruction-level reverse execution for debugging," in *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2002.

[30] T. Akgul, V. J. Mooney III, and S. Pande, "A fast assembly level reverse execution method via dynamic slicing," in *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[31] C. Hou, G. Vulov, D. Quinlan, D. Jefferson, R. Fujimoto, and R. Vuduc, "A new method for program inversion," in *Proceedings of the 21st International Conference on Compiler Construction*, 2012.

[32] C. Zamfir, B. Kasikci, J. Kinder, E. Bugnion, and G. Candea, "Automated debugging for arbitrarily long executions," in *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, 2013.

[33] B. Liblit and A. Aiken, "Building a better backtrace: Techniques for postmortem program analysis," tech. rep., 2002.

[34] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: Locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.

[35] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: A technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[36] C. Cifuentes and A. Fraboulet, "Intraprocedural static slicing of binary executables," in *Proceedings of 13rd the International Conference on Software Maintenance (ICSM)*, 1997.

[37] T. W. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *Proceedings of the 17th International Conference on Compiler Construction (CC)*, 2008.

[38] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, 2010.

Dongliang Mu received the B.E. degree in computer science and technology from Zhengzhou University in 2014. He is currently pursuing the Ph.D. degree in the Department of Computer Science and Technology, Nanjing University. His current research interests span the area of vulnerability reproduction, postmortem program analysis, vulnerability diagnosis, and binary analysis.

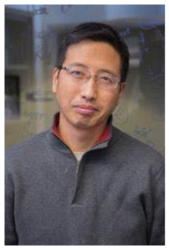
Yunlan Du received the B.E. degree in Nanjing University of Post and Telecommunications in 2017. She is currently pursuing the M.S. degree in the Department of Computer Science and Technology, Nanjing University. Her research interests include system and software security, especially vulnerability detection.

Jianhao Xu received the B.E. degree in computer science and technology from Nanjing University of Science and Technology in 2017. He is currently pursuing the M.S. degree with the Department of Computer Science and Technology, Nanjing University. His current research interests include software and system security.

Jun Xu is an Assistant Professor in the Department of Computer Science at Stevens Institute of Technology. He received his Ph.D. from the College of Information Sciences and Technology at the Pennsylvania State University. His research area covers software security, system security, and malware.

Xingyu Xing is an Assistant Professor in the College of Information Sciences and Technology at the Pennsylvania State University. He earned his Ph.D. in Computer Science from Georgia Tech. His research area covers system security, binary analysis and deep learning.

Bing Mao is a Professor in the Department of Computer Science and Technology at Nanjing University. He received the M.S. and Ph.D. degrees from Nanjing University in 1994 and 1996, respectively, both in computer science. His current research interests include system and software security.



Peng Liu is a Professor of Information Sciences and Technology, founding Director of the Center for Cyber Security, Information Privacy, and Trust, and founding Director of the Cyber Security Lab at Penn State University. His research interests are in all areas of computer and network security.