Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts

Jun Xu, *The Pennsylvania State University;* Dongliang Mu, *Nanjing University;* Xinyu Xing, Peng Liu, and Ping Chen, *The Pennsylvania State University;* Bing Mao, *Nanjing University*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xu-jun

This paper is included in the Proceedings of the
26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

POMP: Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts

Jun Xu[†], Dongliang Mu[‡†], Xinyu Xing[†], Peng Liu[†], Ping Chen[†], and Bing Mao[‡]

[†] *College of Information Sciences and Technology, The Pennsylvania State University*

[‡] *State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University*

{jxx13,dzm77,xxing, pliu, pzc10}@ist.psu.edu, {maobing}@nju.edu.cn

Abstract

While a core dump carries a large amount of information, it barely serves as *informative* debugging aids in locating software faults because it carries information that indicates only a partial chronology of how program reached a crash site. Recently, this situation has been significantly improved. With the emergence of hardware-assisted processor tracing, software developers and security analysts can trace program execution and integrate them into a core dump. In comparison with an ordinary core dump, the new post-crash artifact provides software developers and security analysts with more clues as to a program crash. To use it for failure diagnosis, however, it still requires strenuous manual efforts.

In this work, we propose `POMP`, an automated tool to facilitate the analysis of post-crash artifacts. More specifically, `POMP` introduces a new reverse execution mechanism to construct the data flow that a program followed prior to its crash. By using the data flow, `POMP` then performs backward taint analysis and highlights those program statements that actually contribute to the crash.

To demonstrate its effectiveness in pinpointing program statements truly pertaining to a program crash, we have implemented `POMP` for Linux system on x86-32 platform, and tested it against various program crashes resulting from 31 distinct real-world security vulnerabilities. We show that, `POMP` can accurately and efficiently pinpoint program statements that truly pertain to the crashes, making failure diagnosis significantly convenient.

1 Introduction

Despite the best efforts of software developers, software inevitably contains defects. When they are triggered, a program typically crashes or otherwise terminates abnormally. To track down the root cause of a software crash, software developers and security analysts need to identify those program statements pertaining to the crash, analyze these statements and eventually figure out why a bad value (such as an invalid pointer) was passed to the crash site. In general, this procedure can be significantly facilitated (and even automated) if both control and data flows are given. As such, the research on postmortem program analysis primarily focuses on finding out control and data flows of crashing programs. Of all techniques on postmortem program analysis, record-and-replay (*e.g.,* [10, 12, 14]) and core dump analysis (*e.g.,* [16, 26, 36]) are most common.

Record-and-replay is a technique that typically instruments a program so that one can automatically log non-deterministic events (*i. e.,* the input to a program as well as the memory access interleavings of the threads) and later utilize the log to replay the program deterministically. In theory, this technique would significantly benefit root cause diagnosis of crashing programs because developers and security analysts can fully reconstruct the control and data flows prior to a crash. In practice, it however is not widely adopted due to the requirement of program instrumentation and the high overhead it introduces during normal operations.

In comparison with record-and-reply, core dump analysis is a lightweight technique for the diagnosis of program crashes. It does not require program instrumentation, nor rely upon the log of program execution. Rather, it facilitates program failure diagnosis by using more generic information, *i. e.,* the core dump that an operating system automatically captures every time a process has crashed. However, a core dump provides only a snapshot of the failure, from which core dump analysis techniques can infer only partial control and data flows pertaining to program crashes. Presumably as such, they have not been treated as the first choice for software debugging.

Recently, the advance in hardware-assisted processor tracing significantly ameliorates this situation. With the emergence of Intel PT [6] – a brand new hardware feature in Intel CPUs – software developers and security analysts can trace instructions executed and save them in a circular buffer. At the time of a program crash, an 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.

While Intel PT augments software developers with the ability of obtaining more informative clues as to a software crash, to use it for the root cause diagnosis of software failures, it is still time consuming and requires a lot of manual efforts. As we will discuss in Section 2, a post-crash artifact[1] typically contains a large amount of instructions. Even though it carries execution history that allows one to fully reconstruct the control flow that a crashing program followed – without an automated tool to eliminate those instructions not pertaining to the failure – software developers and security analysts still need to manually examine each instruction in an artifact and identify those that actually contribute to the crash.

To address this problem, recent research [22] has proposed a technical approach to identify program statements that pertain to a software failure. Technically speaking, it combines static program analysis with a cooperative and adaptive form of dynamic program analysis that uses Intel PT. While shown to be effective in facilitating failure diagnosis, particularly those caused by concurrency bugs, this technique is less likely to be effective in analyzing crashes resulting from memory corruption vulnerabilities (e.g. buffer overflow or use after free). This is due to the fact that a memory corruption vulnerability allows an attacker to manipulate the control (or data) flow, whereas the static program analysis heavily relies upon the assumption that program execution does not violate control nor data flow integrity. Given that the technique proposed in [22] needs to track data flow using hardware watchpoints in a collaborative manner, this technique is also less suitable to the situation where program crashes cannot be easily collected in a crowd-sourcing manner.

In this work, we design and develop POMP, a new automated tool that analyzes a post-crash artifact and pinpoints statements pertaining to the crash. Considering that the control flow of a program might be hijacked and static analysis is unreliable, the design of POMP is exclusively on the basis of the information residing in post-crash artifacts. In particular, POMP introduces a reverse execution mechanism which takes as input a post-crash artifact, analyzes the crashing memory and reversely executes the instructions residing in the artifact. With the support of this reverse execution, POMP reconstructs the data flow that a program followed prior to its crash, and then utilizes backward taint analysis to pinpoint the critical instructions leading up to the crash.

The reverse execution proposed in this work is novel. In previous research, the design of reverse execution is under the assumption of the data integrity in crashing memory [16, 37] or heavily relies upon the capability of recording critical objects in memory [7–9, 13]. In this work, considering a software vulnerability might incur memory corruption and object recording imposes overhead on normal operations, we relax this assumption and the ability of data object recording, and introduce a recursive algorithm. To be specific, the algorithm performs the restoration of memory footprints by constructing the data flow prior to the crash. In turn, it also employs recovered memory footprints to improve data flow construction. If needed, the algorithm also verifies memory aliases and ensures data flow construction does not introduce errors or uncertainty. We detail this algorithm in Section 4.

To the best of our knowledge, POMP is the first work that can recover the data flow prior to a program crash. Since POMP relies only upon a post-crash artifact, it is non-intrusive to normal operations and, more importantly, generally applicable to any settings even though crash report collection cannot be performed in a cooperative manner. Last but not least, it should be noted that the impact of this work is not just restricted to analyzing the abnormal program termination caused by memory corruption vulnerabilities. The technique we proposed is generally applicable to program crashes caused by other software bugs, such as dereferencing null pointers. We will demonstrate this capability in Section 6.

In summary, this paper makes the following contributions.

- We designed POMP, a new technique that analyzes post-crash artifacts by reversely executing instructions residing in the artifact.
- We implemented POMP on 32-bit Linux for facilitating software developers (or security analysts) to pinpoint software defects, particularly memory corruption vulnerabilities.
- We demonstrated the effectiveness of POMP in facilitating software debugging by using various post-crash artifacts attributable to 31 distinct real world security vulnerabilities.

The rest of this paper is organized as follows. Section 2 defines the problem scope of our research. Section 3 presents the overview of POMP. Section 4 and 5 describe the design and implementation of POMP in detail. Section 6 demonstrates the utility of POMP. Section 7 summarizes the work most relevant to ours followed by some discussion on POMP in Section 8. Finally, we conclude this work in Section 9.

[1] By a post-crash artifact, without further specification, we mean a core dump including both the snapshot of crashing memory and the instructions executed prior to the crash.

```
 1  void test(void){
 2     ...
 3  }
 4
 5  int child(int *a){
 6     a[0] = 1; // assigning value to var
 7     a[1] = 2; // overflow func
 8     return 0;
 9  }
10
11  int main(){
12     void (*func)(void);
13     int var;
14     func = &test;
15     child(&var);
16     func(); // crash site
17  }
```

Table 1: A toy example with a stack overflow defect.

2 Problem Scope

In this section, we define the problem scope of our research. We first describe our threat model. Then, we discuss why failure diagnosis can be tedious and tough even though a post-crash artifact carries information that allows software developers to fully reconstruct the control flow that a program followed prior to its crash.

2.1 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. Up to and including Linux 2.2, the default action for CPU time limit exceeded, for example, is to terminate the process without a post-crash artifact [3].

As is mentioned above, 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[2]. Recall that the goal of this work is to identify those program statements (*i. e.,* instructions) that actually pertain to the crash. Therefore, we assume the instruction trace logged in an artifact is sufficiently long and the root cause of a program failure is always enclosed. In other words, 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 [19, 27, 39] and an operating system can easily allocate a memory region to store the execution trace from a defect triggered to an actual crash. Since security analysts may not have the access to source code of crashing programs and they can only pinpoint software defects using execution traces left behind crashes, it should be noted that we do not assume the source code of the crashing program is available.

2.2 Challenge

As is mentioned earlier, Intel PT records program execution in a circular buffer. At the time a software defect is triggered and incurs a crash, the circular buffer has generally accumulated a large amount of conditional branches. After the control flow reconstruction from these branches, a full execution trace may carry more than a billion instructions. Even if zooming in the trace from where a fault is triggered to where a crash occurs, a software developer (or security analyst) may confront tens of thousands of instructions. As such, it is tedious and arduous for a software developer to plow through an execution trace to diagnose the root cause of a software failure.

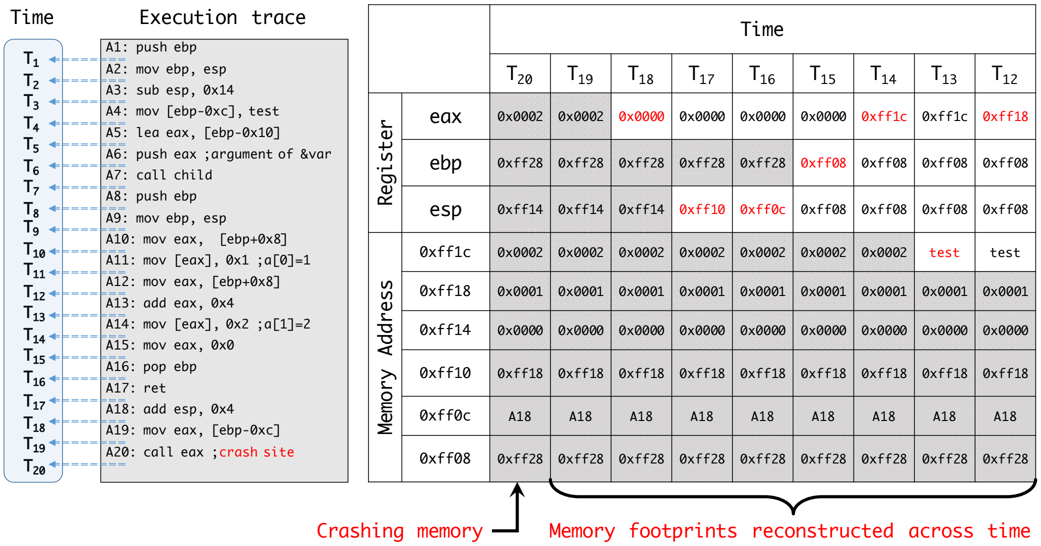
In fact, even though an execution trace is short and concise, it is still challenging for commonly-adopted manual diagnosis strategies (like backward analysis). Here, we detail this challenge using a toy example shown in Table 1. As is shown in the table, the program crashes at `line 16` due to an overflow that occurs at `line 7`. After the crash, an execution trace is left behind in a post-crash artifact shown in Figure 1. In addition to the trace, the artifact captures the state of the crashing memory which is illustrated as the values shown in column T_{20}.

To diagnose the root cause with backward analysis for the program crash shown in Figure 1, a software developer or security analyst typically follows through the execution trace reversely and examines how the bad value in register `eax` was passed to the crash site (*i. e.,* instruction `A20` shown in Figure 1). In this procedure, his effort can be prematurely blocked when his analysis reaches instruction `A19`. In that instruction `mov` overwrote register `eax` and an inverse operation against this instruction lacks information to restore its previous value.

To address this problem, one straightforward solution is to perform forward analysis when backward analysis reaches a non-invertible instruction. Take instruction `A19` for the example. By following a use-define chain, we can construct a data flow. 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`.

While the backward and forward analysis provides security analysts with an effective method to construct data

[2] 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: mov [ebp-0xc], test
A5: lea eax, [ebp-0x10]
A6: push eax ;argument of &var
A7: call child
A8: push ebp
A9: mov ebp, esp
A10: mov eax,  [ebp+0x8]
A11: mov [eax], 0x1 ;a[0]=1
A12: mov eax, [ebp+0x8]
A13: add eax, 0x4
A14: mov [eax], 0x2 ;a[1]=2
A15: mov eax, 0x0
A16: pop ebp
A17: ret
A18: add esp, 0x4
A19: mov eax, [ebp-0xc]
A20: call eax ;crash site
```

		Time								
		T_{20}	T_{19}	T_{18}	T_{17}	T_{16}	T_{15}	T_{14}	T_{13}	T_{12}
Register	eax	0x0002	0x0002	0x0000	0x0000	0x0000	0x0000	0xff1c	0xff1c	0xff18
	ebp	0xff28	0xff28	0xff28	0xff28	0xff28	0xff08	0xff08	0xff08	0xff08
	esp	0xff14	0xff14	0xff14	0xff10	0xff0c	0xff08	0xff08	0xff08	0xff08
Memory Address	0xff1c	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	test	test
	0xff18	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001
	0xff14	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
	0xff10	0xff18	0xff18	0xff18	0xff18	0xff18	0xff18	0xff18	0xff18	0xff18
	0xff0c	A18	A18	A18	A18	A18	A18	A18	A18	A18
	0xff08	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28

Figure 1: A post-crash artifact along with the memory footprints 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.

flows, this is not sufficient for completing program failure diagnosis. Again, take for example the execution trace shown in Figure 1. When backward analysis passes through instruction A15 and reaches instruction A14, through forward analysis, a security analyst can quickly discover that the value in register eax after the execution of A14 is dependent upon both instruction A12 and A13. As a result, an instinctive reaction is to retrieve the value stored in the memory region specified by [ebp+0x8] shown in instruction A12. However, memory indicated by [ebp+0x8] and [eax] shown in instruction A14 might be alias of each other. Without an approach to resolve memory alias, one cannot determine if the definition in instruction A14 interrupts the data flow from instructions A12 and A13. Thus, program failure diagnosis has to discontinue without an outcome.

3 Overview

In this section, we first describe the objective of this research. Then, we discuss our design principle followed by the basic idea on how POMP performs postmortem program analysis.

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. Given a post-crash artifact containing an execution trace carrying a large amount of instructions that a program has executed prior to its crash, however, any instructions in the trace can be potentially attributable to the crash. As we have shown in the section above, it is tedious and tough for software developers (or security analysts) to dig through the trace and pinpoint the root cause of a program crash. Therefore, the objective of this work is to identify only those instructions that truly contribute to the crash. In other words, given a post-crash artifact, our goal is to highlight and present to software developers (or security analysts) the minimum set of instructions that contribute to a program crash. Here, our hypothesis is that the achievement of this goal can significantly reduce the manual efforts of finding out the root cause of a software failure.

3.2 Design Principle

To accomplish the aforementioned objective, we design POMP to perform postmortem analysis on binaries – though in principle this can be done on a source code level – in that this design principle can provide software developers and security analysts with the following benefits. Without having POMP tie to a set of programs written in a particular programming language, our design principle first allows software developers to employ a single tool to analyze the crashes of programs written in various language (*e.g.,* assembly code, C/C++ or JavaScript). Second, our design choice eliminates the complication introduced by the translation between source code and binaries in that a post-crash artifact carries an execution trace in binaries which can be directly consumed by analysis at the binary level. Third, with the choice of our design, POMP can be generally applied to software failure triage or categorization in which a post-crash artifact is the only resource for analysis and the source code of a crashing program is typically not available [16, 18].

3.3 Technical Approach

As is mentioned earlier in Section 1, it is significantly convenient to identify the instructions pertaining to a program crash if software developers and security analysts can obtain the control and data flows that a program followed prior to its crash.

We rely on Intel PT to trace the control flow of a program and integrate it into the post-crash artifact. PT is a low-overhead hardware feature in recent Intel processors (*e.g.,* Skylake series). It works by capturing information about software execution on each hardware thread [6]. The captured information is orgranized in different types of data packets. Packets about program flow encodes the transfers of control flow (*e.g.,* targets of indirect branches and taken/not-taken indications of conditional direct branches). With the control flow transfers and the program binaries, one is able to fully reconstruct the trace of executed instructions. Details of our configuration and use with PT are presented in Section 5.

Since a post-crash artifact has already carried the control flow that a crashing program followed, the main focus is to reconstruct the data flow from the post-crash artifact that a crashing program left behind.

To reconstruct the data flow pertaining to a program failure, POMP introduces a reverse execution mechanism to restore the memory footprints of a crashing program. This is due to the fact that the data flow can be easily derived if machine states prior to a program crash are all available. In the following, we briefly describe how to recover memory footprints and build a data flow through reverse execution, and how to utilize that data flow to refine instructions that truly pertain to a program crash.

Our reverse execution mechanism is an extension of the aforementioned forward-and-backward analysis. Not only does it automate the forward-and-backward analysis, making the inverse operations for instructions effortless, but also automatically verifies memory aliases and ensures an inverse operation does not introduce errors or uncertainty.

With this reverse execution mechanism, POMP can easily restore the machine states prior to the execution of each instruction. Here, we illustrate this with the example shown in Figure 1. After reverse execution completes the inverse operation for instruction A19 through the aforementioned forward and backward analysis, it can easily restore the value in register eax and thus the memory footprint prior to the execution of A19 (see memory footprint at time T_{18}). With this memory footprint, the memory footprint prior to instruction A18 can be easily recovered because arithmetical instructions do not introduce non-invertible effects upon memory (see the memory footprint at time T_{17}).

Since instruction A17 can be treated as mov eip, [esp] and then add esp, 0x4, and instruction A16 is equivalent to mov ebp, [esp] and then add esp, 0x4, reverse execution can further restore memory footprints prior to their execution by following the scheme of how it handles mov and arithmetical instructions above. In Figure 1, we illustrate the memory footprints prior to the execution of both instructions.

Recall that performing an inverse operation for instruction A15, forward and backward analysis cannot determine whether the use of [ebp+0x8] specified in instruction A12 can reach the site prior to the execution of instruction A15 because [eax] in A14 and [ebp+0x8] in A12 might just be different symbolic names that access data in the same memory location.

To address this issue, one instinctive reaction is to use the value-set analysis algorithm proposed in [11]. However, value-set analysis assumes the execution complies with standard compilation rules. When memory corruption happens and leads to a crash, these rules are typically violated and, therefore, value-set analysis is very likely to be error-prone. In addition, value-set analysis produces less precise information, not suitable for reverse execution to verify memory aliases. In this work, we employ a hypothesis test to verify possible memory aliases. To be specific, our reverse execution creates 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 shown in Figure 1. Now, reverse execution can create two hypotheses, one assuming [eax] and [ebp+0x8] are aliases of each other while the other assuming the opposite. For the first hypothesis, after performing the inverse operation for instruction A15, the information carried by the memory footprint at T_{14} would have three constraints, including $eax = ebp + 0x8$, $eax = [ebp + 0x8] + 0x4$ and $[eax] = 0x2$. For the second hypothesis, the constraint set would include $eax \neq ebp + 0x8$, $eax = [ebp + 0x8] + 0x4$ and $[eax] = 0x2$. 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. In this way, reverse execution can efficiently and accurately recover the memory footprint at time T_{14}. After the memory footprint recovery at T_{14}, reverse execution can further restore earlier memory footprints using the scheme we discussed above, and Figure 1 illustrates part of these memory footprints.

With memory footprints recovered, software developers and security analysts can easily derive the correspond-

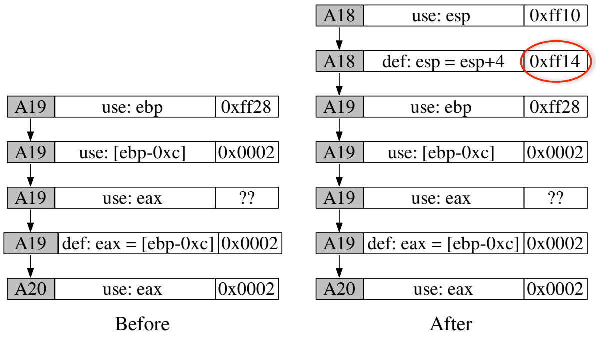


Figure 2: A use-define chain before and after appending new relations derived from instruction `A18`. Each node is partitioned into three cells. From left to right, the cells carry instruction ID, definition (or use) specification and the value of the variable. Note that symbol ?? indicates the value of that variable is unknown.

ing data flow and thus pinpoint instructions that truly contribute to a crash. In our work, `POMP` automates this procedure by using backward taint analysis. To illustrate this, we continue the aforementioned example and take the memory footprints shown in Figure 1. As is described earlier, in this case, the bad value in register `eax` was passed through instruction `A19` which copies the bad value from memory `[ebp-0xC]` to register `eax`. By examining the memory footprints restored, `POMP` can easily find out that the memory indicated by `[ebp-0xC]` 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 `A19` and `A14`, and deems they are truly attributable to the crash. We elaborate on the backward taint analysis in Section 4.

4 Design

Looking closely into the example above, we refine an algorithm to perform reverse execution and memory footprint recovery. In the following, we elaborate on this algorithm followed by the design detail of our backward taint analysis.

4.1 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.

4.1.1 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 `A20` and `A19` shown in Figure 1, `POMP` extracts the use and definition from instruction `A18` and links them to the head of the chain (see Figure 2).

As we can observe from 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 2).

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, we mean the variable satisfies one of the following conditions – ① the definition (or use) of that variable could reach the end of the chain without any other intervening definitions; ② it could 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 2. 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, none of the definitions and uses on the chain can benefit from this propagation. After the completion of this propagation, our algorithm further appends use `use:esp` and repeats this process. Slightly different from the process for definition `def:esp=esp+4`, for this use, variable `esp` is not resolvable through the aforementioned reachability examination. Therefore, our algorithm derives the value of `esp` from the semantics of instruction `A18`

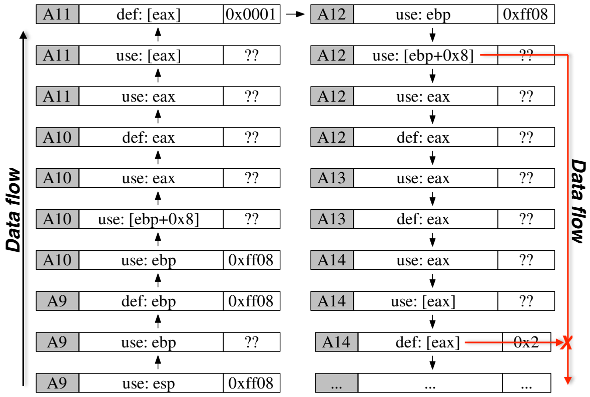


Figure 3: A use-define chain with one intervening tag conservatively placed. The tag blocks the propagation of some data flows. Note that ✗ represents the block of a data flow.

(*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. Take for example instruction `A19` and dummy instruction sequence `cmp eax, ebx;` ⇒ `ja target;` ⇒ `inst_at_target` . Our algorithm extracts equality constraint `eax=[ebp-0xc]` and inequality constraint `eax>ebx`, respectively. 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 encounters 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 region but the address of that region cannot be resolved by using the information on the chain. For example, instruction `A14` shown in Figure 1 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 3, 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.

As we can observe from the example shown in Figure 3, 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, given that memory indicated by `[ebp+0x08]` and `[eax]` might be an alias of each other, definition `A14 def:[eax]` may block the reachability of `A12 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 principle ensures that our algorithm does not introduce errors to memory footprint recovery.

The above forward-and-backward analysis is mainly designed to discover the use-define reltaions. Other techniques, such as static program slicing [34], 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. Additionally, our analysis conservatively deals with memory aliases and verifies them in an error-free manner. This is different from previous techniques that typically leverage less rigorous methods (*e.g.,* value-set analysis). More details about how we resolve memory alias are presented in the next section.

4.1.2 Memory Alias Verification

While the aforementioned design principle prevents introducing errors to memory footprint recovery, this conservative strategy hinders data flow construction and limits the capability of resolving variables (see the flow block and non-recoverable variables shown in Figure 3). As a result, the second step of our algorithm is to minimize the side effect introduced by the aforementioned strategy.

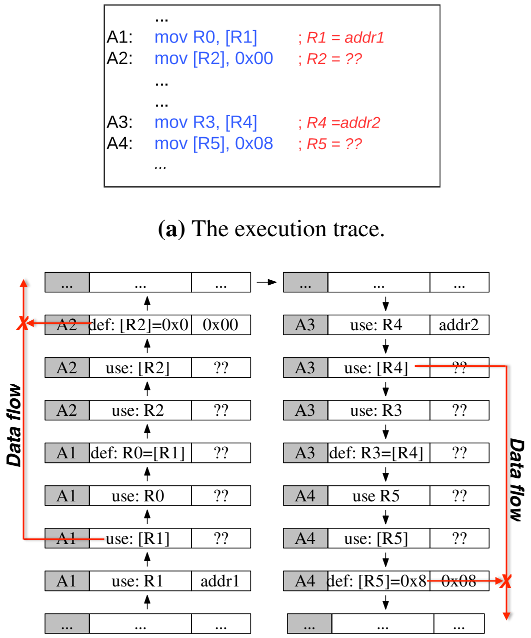
Since the conservative design above roots in "undecidable" memory alias, the way we tackle the problem is to introduce a hypothesis test mechanism that examines if a pair of symbolic names points to the same memory location. More specifically, given a pair of symbolic names, this mechanism makes two hypotheses, one assuming they are alias of each other and the other assuming the opposite. Based on the hypotheses, our algorithm adjusts the use-define chain as well as constraints accordingly. For example, by assuming `[eax]` is not aliased to `[ebp+0x8]`, our algorithm extracts inequility constraint `eax≠ebp+0x8` and releases the block shown in Figure 3, making `A12 use:[ebp+0x8]` further propagated.

During the propagation, our algorithm walks through each of the nodes on the chain and examines if the newly propagated data flow results in conflicts. Typically, there are two types of conflicts. The most common is inconsistence data dependency in which constraints mismatch the data propagated from above (*e.g.,* the example discussed in Section 3). In addition to the conflict commonly observed, another type of conflict 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. For example, given a use-define chain established under a certain hypothesis, the walk-through discovers that a register carries an invalid address and that invalid value should have the crashing program terminate at a site ahead of its actual crash site.

It is indisputable that 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. However, if none of these hypotheses produce constraint conflicts, this implies that there is a lack of evidence against our hypothesis test. Once this situation appears, our algorithm holds the current hypothesis and performs an additional hypothesis test. The reason is that a new hypothesis test may help remove an additional intervening tag conservatively placed at the first step, and thus provides the holding test with more informative evidence to reject hypotheses accordingly.

To illustrate this, we take a simple example shown in Figure 4. After the completion of the first step, we assume that our algorithm conservatively treats `A2 def:[`R_2`]` and `A4 def:[`R_5`]` as intervening tags which hinder data flow propagation. Following the procedure discussed above, we reversely analyze the trace and make a hypothesis, *i. e.,* `[`R_4`]` and `[`R_5`]` are not alias. With this hypothesis, the data flow between the intervening tags can propagate through, and our algorithm can examine conflicts accordingly. Assume that the newly propagated data flow is insufficient for rejecting our hypothesis. Our algorithm holds the current hypothesis and makes an additional hypothesis, *i. e.,* `[`R_1`]` and `[`R_2`]` are not alias of each other. With this new hypothesis, more data flows pass through and our algorithm obtains more information that potentially helps reject hypotheses. It should be noted that if any of the hypotheses fail to reject, our algorithm preserves the intervening tags conservatively placed at the first step.

It is not difficult to spot that our hypothesis test can be easily extended as a recursive procedure which makes more hypotheses until they can be rejected. However, a recursive hypothesis test introduces computation complexity exponentially. In the worse case, when performing execution reversely, the inverse operation of each instruction may require alias verification and each verification may require further alias examination. When this situation appears, the algorithm above becomes an impractical solution. As such, this work empirically forces a hypothesis test to follow at most a recurssion depth of two. As we will show in Section 6, this setting allows us to perform reverse execution not only in an efficient but also relatively effective manner.



```
     ...
A1:  mov R0, [R1]      ; R1 = addr1
A2:  mov [R2], 0x00    ; R2 = ??
     ...
     ...
A3:  mov R3, [R4]      ; R4 =addr2
A4:  mov [R5], 0x08    ; R5 = ??
     ...
```

(a) The execution trace.

(b) The use-define chain.

Figure 4: A dummy use-define chain and execution trace with two pairs of memory aliases. Note that $R_0, R_1, \cdots R_5$ represent registers in which the values of R_2 and R_5 are unknown. Note that ✗ represents the block of a data flow.

4.1.3 Discussion

During the execution of a program, it might invoke a system call, which traps execution into kernel space. As we will discuss in Section 6, we do not set Intel PT to trace execution in kernel space. As a result, intuition suggests that the loss of execution tracing may introduce problems to our reverse execution. However, in practice, a majority of system calls do not incur modification to registers and memory in user space. Thus, our reverse execution can simply ignore the inverse operations for those system calls. For system calls that potentially influence the memory footprints of a crashing program, our reverse execution handles them as follows.

In general, a system call can only influence memory footprints if it manipulates register values stored by the crashing program or touches the memory region in user space. As a result, we treat system calls in different manners. For system calls that may influence a register holding a value for a crashing program, our algorithm simply introduces a definition on the use-define chain. For example, system call read overwrites register eax to hold its return value, and our algorithm appends definition def:eax=? to the use-define chain accordingly. Regarding the system calls that manipulate the memory content in user space (*e.g.,* write and recv), our algorithm checks the memory regions influenced by that call. To be specific, it attempts to identify the starting address as well as the size of that memory region by using the instructions executed prior to that call. This is due to the fact that the starting address and size are typically indicated by arguments which are handled by those instructions prior to the call. Following this procedure, if our algorithm identifies the size of that memory region, it appends definitions to the chain accordingly. Otherwise, our algorithm treats that system call as an intervening tag which blocks the propagation through that call[3]. The reason behind this is that a non-deterministic memory region can potentially overlap with any memory regions in user space.

4.2 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 the 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. Take the example shown in Figure 1. POMP treats register eax as a sink in that the program crash results from retrieving an invalid instruction from the address held by register eax.

With a sink identified, POMP taints the sink and performs taint propagation backward. In the procedure of this backward propagation, POMP looks up the aforementioned use-define chain and identifies the definition of the taint variable. The criteria of this identification is to ensure the definition could reach the taint variable without any other intervening definitions. Continue the example above. With sink eax serving as the initial taint variable, POMP selects A19 def:eax=[ebp-0xc] on the chain because this definition can reach taint variable eax without intervention.

From the definition identified, POMP parses that definition and passes the taint to new variables. Since any variables enclosed in a definition could potentially cause the corruption of the taint variable, the variables which POMP selects and passes the taint to include all operands, base and index registers (if available). For example, by parsing definition A19 def:eax=[ebp-0xc], POMP identifies variables ebp and [ebp-0xc], and passes the taint to both of them. It is not difficult to note that such a taint propagation strategy can guarantee POMP does not miss the root cause of a program crash though it over-taints some variables that do not actually contribute to the crash. In Section 6, we evaluate and discuss the effect of the over-tainting.

When passing a taint to a variable indicated by a memory access (*e.g.,* [R_0]), it should be noted that POMP may not be able to identify the address corresponding to the memory (*e.g.,* unknown R_0 for variable [R_0]). Once this situation appears, therefore, POMP halts the taint propagation for that variable because the taint can be potentially propagated to any variables with a definition in the form of def:[R_i] (where R_i is a register).

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 taint 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 hypothesis test and examines if both variables share the same address. Ideally, we would like to resolve the unknown address through a hypothesis test so that POMP can pass that taint accordingly. However, in practice, the hypothesis test may fail to reject. 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 root cause.

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 two major components – ① a sub-system that implements the aforementioned reverse execution and backward taint analysis and ② a sub-system that traces program execution with Intel PT. In total, our implementation carries about 22,000 lines of C code which we will make publicly available at https://github.com/junxzm1990/pomp.git. In the following, we present some important implementation details.

[3] Note that an intervening tag placed by a system call blocks only definitions and uses in which a variable represents a memory access (*e.g.,* def:[eax] or use:[ebp]).

Following the design description above, we implemented 65 distinct instruction handlers to perform reverse execution and backward taint analysis. Along with these handlers, we also built core dump and instruction parsers on the basis of libelf [2] and libdisasm [1], 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 [5, 17].

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. Considering the interrupt may have a skid, resulting in a potential loss in PT packets, we further allocated a 2 MB physical memory buffer to hold those packets that might be potentially discarded. In the ToPA, we introduced an additional entry to refer this buffer.

At the hardware level, Intel PT lacks the capability of distinguishing threads within each process. As a result, we also intercepted the context switch. With this, our system is able to examine the threads switched in and out, and stores PT packets for threads individually. To be specific, for each thread that software developers and security analysts are interested in, we allocated a 32MB circular buffer in its user space. Every time a thread is switched out, we migrated PT packets stored in the aforementioned physical memory buffers to the corresponding circular buffer in user space. After migration, we also reset the corresponding registers and make sure the physical memory buffers can be used for holding packets for other threads of interest. Note that our empirical experiment indicates the aforementioned 16 MB buffer cannot be fully occupied between consecutive context switch, and POMP does not have the difficulty in holding all the packets between the switch.

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, not only can software developers and security analysts select which processes to trace but also configure the size of the circular buffer in a convenient manner.

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 efficiency and effectiveness of POMP, and discuss those crashes that POMP fails to handle properly.

6.1 Setup

To demonstrate the utility of POMP, we selected 28 programs and benchmarked POMP with their crashes resulting from 31 real-world PoCs obtained from Offensive Security Exploit Database Archive [4]. Table 2 shows these crashing programs and summarizes the corresponding vulnerabilities. As we can observe, the programs selected 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 memory corruption vulnerabilities (*i. e.,* stack and heap overflow) but also 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.

Among the 32 PoCs, 11 of them perform code injection (*e.g.,* nginx-1.4.0), one does return-to-libc attack (aireplay-ng-1.2b3), and another one exploits via return-oriented-programming (mcrypt-2.5.8). These exploits crashed the vulnerable program either because they did not consider the dynamics in the execution environments (*e.g.,* ASLR) or they mistakenly polluted critical data (*e.g.,* pointers) before they took over the control flow. The remaining 18 PoCs are created to simply trigger the defects, such as overflowing a stack buffer with a large amount of random characters (*e.g.,* BinUtils-2.15) or causing the execution to use a null pointer (*e.g.,* gdb-7.5.1). Crashes caused by these PoCs are similar to those occured during random exercises.

6.2 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

Program		***Vulnerability***		***Diagnose Results***						
Name	Size(LoC)	CVE-ID	Type	Trace length	Size of mem (MB)	# of taint	Ground truth	Mem addr unknown	Root cause	Time
coreutils-8.4	138135	2013-0222	Stack overflow	50	56.61	3	2	1	✓	1 sec
coreutils-8.4	138135	2013-0223	Stack overflow	90	59.66	2	2	0	✓	1 sec
coreutils-8.4	138135	2013-0221	Stack overflow	92	120.95	3	2	0	✓	1 sec
mcrypt-2.5.8	37439	2012-4409	Stack overflow	315	0.59	3	2	3	✓	3 sec
BinUtils-2.15	697354	2006-2362	Stack overflow	867	0.37	16	7	0	✓	1 sec
unrtf-0.19.3	5039	NA	Stack overflow	895	0.34	7	4	10	✓	1 min
psutils-p17	1736	NA	Stack overflow	3123	0.34	7	3	28	✓	4 min
stftp-1.1.0	1559	NA	Stack overflow	3651	0.39	29	6	15	✓	4 min
nasm-0.98.38	33553	2004-1287	Stack overflow	4064	0.58	3	2	4	✓	44 sec
libpng-1.2.5	33681	2004-0597	Stack overflow	6026	0.35	6	2	86	✓	5 min
putty-0.66	90165	2016-2563	Stack overflow	7338	0.45	4	2	21	✓	30 min
Unalz-0.52	8546	2005-3862	Stack overflow	10905	0.40	14	10	7	✓	30 sec
LaTeX2rtf-1.9	14473	2004-2167	Stack overflow	17056	0.37	11	5	122	✓	8 min
aireplay-ng-1.2b3	62656	2014-8322	Stack overflow	18569	0.59	2	2	223	✗	7 min
corehttp-0.5.3a	914	2007-4060	Stack overflow	25385	0.32	19	6	0	✓	52 min
gas-2.12	595504	2005-4807	Stack overflow	25713	4.17	3	2	346	✓	40 min
abc2mtex-1.6.1	4052	NA	Stack overflow	29521	0.33	12	2	12	✓	1 min
LibSMI-0.4.8	80461	2010-2891	Stack overflow	50787	0.33	46	5	730	✓	4 sec
gif2png-2.5.2	1331	2009-5018	Stack overflow	70854	0.51	49	4	396	✓	46 min
O3read-0.03	932	2004-1288	Stack overflow	78244	0.32	7	2	20	✓	15 min
unrar-3.9.3	17575	NA	Stack overflow	102200	2.43	33	5	1033	✓	6 hour
nullhttp-0.5.0	1849	2002-1496	Heap overflow	141	0.54	3	2	0	✓	1 sec
inetutils-1.8	98941	NA	Heap overflow	28720	0.40	237	7	111	✓	14 min
nginx-1.4.0	100255	2013-2028	Integer overflow	158	0.62	11	4	0	✓	1 sec
Python-2.2	416060	2007-4965	Integer overflow	3426	0.89	31	7	117	✓	3 min
0verkill-0.16	16361	2006-2971	Integer overflow	10494	4.27	1	NA	0	✗	2 sec
openjpeg-2.1.1	169538	2016-7445	Null pointer	67	0.37	10	5	5	✓	1 sec
gdb-7.5.1	1651764	NA	Null pointer	2009	2.94	23	2	79	✓	1sec
podofo-0.9.4	60147	2017-5854	Null pointer	42165	0.65	7	4	80	✓	2 min
Python-2.7	906829	NA	Use-after-free	551	2.14	6	1	0	✓	0.17 sec
poppler-0.8.4	183535	2008-2950	Invalid free	672	1.39	16	4	0	✓	13 sec

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. Size of mem shows the size of memory used by the crashed program (with code sections excluded). # of taint and Ground truth describe the lines of instructions automatically pinpointed and manually identified, respectively. Mem addr unknown illustrates the amount of memory locations, the addresses of which are unresolvable.

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 if POMP could significantly prune the execution trace that software developers (or security analysts) have to manually examine.

In order to evaluate the efficiency of POMP, we recorded the time it took when spotting the instructions that truly pertain to each program crash. For each test case, we also logged the instructions that POMP reversely executed in that this allows us to study the relation between efficiency and the amount of instructions reversely executed.

Considering pinpointing a root cause does not require reversely executing the entire trace recorded by Intel PT, it is worth of noting 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.

6.3 Experimental Results

We show our experimental results in Table 2. Except for test cases 0verkill and aireplay-ng, we observe, every root cause is included in a set of instructions that POMP pinpointed. Through a comparison mentioned above, we also observe each set encloses the corresponding instructions we manually identified (*i. e.,* ground truth). These observations indicate that POMP is effective in locating instructions that truly contribute to program crashes.

In comparison with instructions that POMP needs to reversely execute, we observe, the instructions eventually tainted are significantly less. For example, backward analysis needs to examine 10,905 instructions in order to pinpoint the root cause for crashing program Unalz, whereas POMP highlights only 14 instructions among which half of them truly pertain to the crash. Given that backward taint analysis mimics how a software developer (or security analyst) typically diagnoses the root cause of a program failure, this observation indicates that POMP has a great potential to reduce manual efforts in failure diagnosis.

Except for test case coreutils, an instruction set produced by POMP generally carries a certain amount of instructions that do not actually contribute to crashes. Again, take Unalz for example. POMP over-tainted 7 instructions and included them in the instruction set it identified. In the usage of POMP, while this implies a software developer needs to devote additional energies to those instructions not pertaining to a crash, this does not mean that POMP is less capable of finding out instructions truly pertaining to a crash. In fact, compared with hundreds and even thousands of instructions that one had to manually walk through in failure diagnosis, the additional effort imposed by over-tainting is minimal and negligible.

Recall that in order to capture a root cause, the design of POMP taints all variables that possibly contribute to the propagation of a bad value. As our backward taint analysis increasingly traverses instructions, it is not difficult to imagine that, an increasing number of variables might be tainted which causes instructions corresponding to these variables are treated as those truly pertaining to program crashes. As such, we generally observe more instructions over-tainted for those test cases, where POMP needs to reversely execute more instructions in order to cover the root causes of their failures.

As we discuss in Section 4, ideally, POMP can employ a recursive hypothesis test to perform inverse operations for instructions that carry unknown memory access. Due to the concern of computation complexity, however, we limit the recursion in at most two depths. As such, reverse execution leaves behind a certain amount of unresolvable memory. In Table 2, we illustrate the amount of memory the addresses of which remain unresolvable even after a 2-depth hypothesis test has been performed. Surprisingly, we discover POMP can still effectively spot instructions pertaining to program crashes even though it fails to recover a certain amount of memory. This implies that our design reasonably balances the utility of POMP as well as its computation complexity.

Intuition suggests that the amount of memory unresolvable should correlate with the number of instructions that POMP reversely executes. This is because the effect of an unresolvable memory might be propagated as more instructions are involved in reverse execution. While this is generally true, an observation from test case corehttp indicates a substantially long execution trace does not always necessarily amplify the influence of unknown memory access. With more instructions reversely executed, POMP may obtain more evidence to reject the hypotheses that it fail to determine, making unknown memory access resolvable. With this in mind, we speculate POMP is not only effective in facilitating failure diagnosis perhaps also helpful for executing substantially long traces reversely. As a future work, we will therefore explore this capability in different contexts.

In Table 2, we also illustrate the amount of time that POMP took in the process of reverse execution and backward taint analysis. We can easily observe POMP typically completes its computation in minutes and the time it took is generally proportional to the number of instructions that POMP needs to reversely execute. The reason behind this observation is straightforward. When reverse execution processes more instructions, it typically encounters more memory aliases. In verifying memory aliases, POMP needs to perform hypothesis tests which are slightly computation-intensive and time-consuming.

With regard to test case aireplay-ng in which POMP fails to facilitate failure diagnosis, we look closely to instructions tainted as well as those reversely executed. Prior to the crash of aireplay-ng, we discover the program invoked system call sys_read which writes a data chunk to a certain memory region. Since both the size of the data chunk and the address of the memory are specified in registers, which reverse execution fails to restore, POMP treats sys_read as a “super” intervening tag that blocks the propagation of many definitions, making the output of POMP less informative to failure diagnosis.

Different from aireplay-ng, the failure for 0verkill results from an insufficient PT log. As is specified in Table 2, the vulnerability corresponding to this case is an integer overflow. To trigger this security loophole, the PoC used in our experiment aggressively accumulates an integer variable which makes a PT log full of arithmetic computation instructions but not the instruction corresponding to the root cause. As such, we observe POMP can taint only one instruction pertaining to the crash. We believe this situation can be easily resolved if a software developer (or security analyst) can enlarge the capacity of the PT buffer.

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. 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 [7–9, 13] 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. For example, the work described in [7–9] is mainly based on regenerating a previous program state. When state regeneration is not possible, however, it recovers a program state by state saving.

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` [21] to instrument C++ program in a way that it can store program states for reverse execution. Similarly, Sauciuc and Necula [30] proposed to use an SMT solver to navigate an execution trace and restore data values. Depending on how the solver performs on constraint sets corresponding to multiple test runs, the technique proposed automatically determines where to instrument the code to save intermediate values and facilitate 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 [16] and [37], new reverse execution mechanisms are designed in which the techniques proposed reversely analyzes code and then utilizes 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 corrupts 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. Over the past decades, there is a rich collection of literature on using program analysis techniques along with crash reports to identify faults in software (*e.g.,* [15, 20, 24, 25, 28, 29, 32, 38]). These existing techniques are designed to identify some specific software defects. In adversarial settings, an attacker exploits a variety of software defects and thus they cannot be used to analyze a program crash caused by a security defect such as buffer overflow or unsafe dangling pointer. For example, Manevich *et al.* [24] proposed to use static backward analysis to reconstruct execution traces from a crash point and thus spot software defects, particularly typestate errors [33]. Similarly, Strom and Yellin [32] defined a partially path-sensitive backward dataflow analysis for checking typestate properties, specifically uninitialized variables. While demonstrated to be effective, these two studies only focus on specific typestate problems.

Liblit *et al.* proposed a backward analysis technique for crash analysis [23]. To be more specific, they introduce 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. In addition, they discussed how to narrow down the set of possible execution paths using a wide variety of post-crash artifacts, such as stack traces. As is mentioned earlier, memory information might be corrupted when attackers exploit a program. The technique described in [23] 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 do not infer program execution paths through the stack traces recovered from memory potentially corrupted. Rather, our approach identifies the root cause of software failures by reversely executing 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 cause of software failures. Of all the works along this line, the most typical ones include `CrashLocator` [35], `!analyze` [18] and `RETracer` [16] 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` [22], 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.* [36] 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 as POMP– 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. While it has been shown that CREDAL is able to assist software developers (or security analysts) in tracking down a memory corruption vulnerability, in most cases, it still requires significant manual efforts for locating a memory corruption vulnerability in a crash for the reasons that the mismatch in variable values may be overwritten or the code fragments corresponding to mismatch may not include the root cause of the software crash. In this work, POMP precisely pinpoints the vulnerability by utilizing the memory footprints recovered from reverse execution.

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. Therefore, we assume the root cause of the crash is enclosed within the instructions executed by that thread and other threads do not intervene the execution of that thread prior to its crash. In practice, this assumption however may not hold, and the information held in a post-crash artifact may not be sufficient and even misleading for root cause diagnosis.

While this multi-thread issue indeed limits the capability of a security analyst using POMP to pinpoint the root cause of a program crash, this does not mean the failure of POMP nor significantly downgrades the utility of POMP because of the following. First, a prior study [31] has already indicated that a large fraction of software crashes involves only the crashing thread. Thus, we believe POMP is still beneficial for software failure diagnosis. Second, the failure of POMP roots in incomplete execution tracing. Therefore, we believe, by simply augmenting our process tracing with the capability of recording the timing of execution, POMP can synthesize a *complete* execution trace, making POMP working properly. As part of the future work, we will integrate this extension into the next version of POMP.

Just-in-Time native code. Intel PT records the addresses of branching instructions executed. Using these addresses as index, POMP retrieves instructions from executable and library files. However, a program may utilize Just-in-Time (JIT) compilation in which binary code is generated on the fly. For programs assembled with this JIT functionality (*e.g.,* JavaScript engine), POMP is less likely to be effective, especially when a post-crash artifact fails to capture the JIT native code mapped into memory.

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. Note that this extension does not require any re-engineering of reverse execution and backward taint analysis because the limitation to JIT native code also results from incomplete execution tracing (*i. e.,* failing to reconstruct all the instructions executed prior to a program crash).

9 Conclusion

In this paper, we develop POMP on Linux system to analyze post-crash artifacts. 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 of POMP using the real-world program crashes pertaining to 31 software vulnerabilities. We showed that POMP can reversely reconstruct the memory footprints of a crashing program and accurately identify the program statements (*i. e.,* , instructions) that truly contribute to the crash. Following this finding, we safely conclude POMP can significantly downsize the program statements that a software developer (or security analyst) needs to manually examine.

10 Acknowledgments

We thank the anonymous reviewers for their helpful feedback and our shepherd, Andrea Lanzi, for his valuable comments on revision of this paper. This work was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1422594, NSF CNS-1505664, ONR N00014-16-1-2265, ARO W911NF-15-1-0576, and Chinese National Natural Science Foundation 61272078.

References

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

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

[3] Linux programmer's manual. http://man7.org/linux/man-pages/man7/signal.7.html.

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

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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *cc*, pages 5–23, 2004.

[12] 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.

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

[14] 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.

[15] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[16] 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.

[17] 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*, pages 337–340. Springer, 2008.

[18] 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.

[19] W. Gu, Z. Kalbarczyk, R. K. Iyer, Z.-Y. Yang, et al. Characterization of linux kernel behavior under errors. In *DSN*, volume 3, pages 22–25, 2003.

[20] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[21] 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.

[22] 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.

[23] B. Liblit and A. Aiken. Building a better backtrace: Techniques for postmortem program analysis. Technical report, 2002.

[24] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. Pse: Explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, 2004.

[25] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

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

[27] 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*, volume 39, pages 235–248. ACM, 2005.

[28] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, 2003.

[29] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization.

In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[30] R. Sauciuc and G. Necula. Reverse execution with constraint solving. Technical report, EECS Department, University of California, Berkeley, 2011.

[31] A. SchrÃűter, 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.

[32] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transaction Software Engineering*, 1993.

[33] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transaction Software Engineering*, 1986.

[34] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[35] 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.

[36] 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.

[37] 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.

[38] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002.

[39] 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*, volume 46, pages 251–264. ACM, 2011.